Podstawy Java – dzień 1

v3.1

Plan

- Powtórzenie z preworku
- Metody
- Tablice

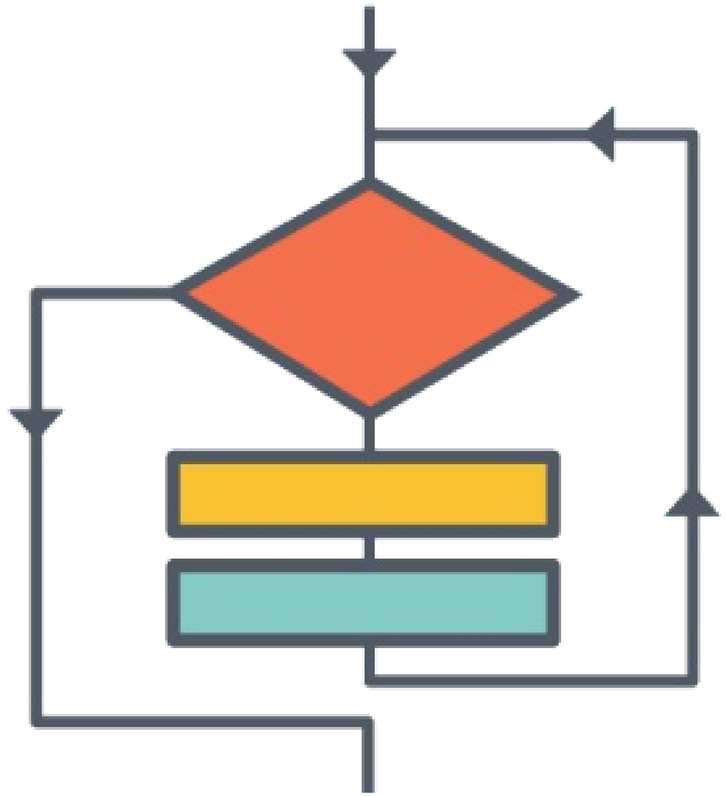
Powtórzenie z preworku

Co to jest algorytm?

Algorytm – skończony ciąg jasno zdefiniowanych czynności koniecznych do wykonania pewnego rodzaju zadań.

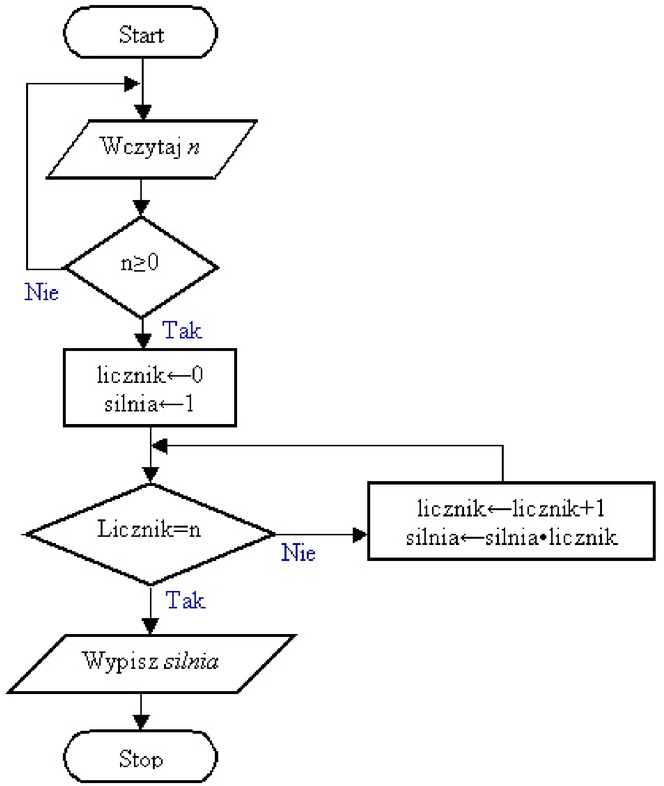
Gdzie na co dzień spotykamy się z algorytmami?

- W kuchni – wszystkie przepisy kulinarne.
- Na drogach – zasady ruchu drogowego.
- Praktycznie w każdej dziedzinie naszego życia (ubieranie się, wiązanie butów, czesanie warkocza itd.).



Schemat blokowy (flowchart)

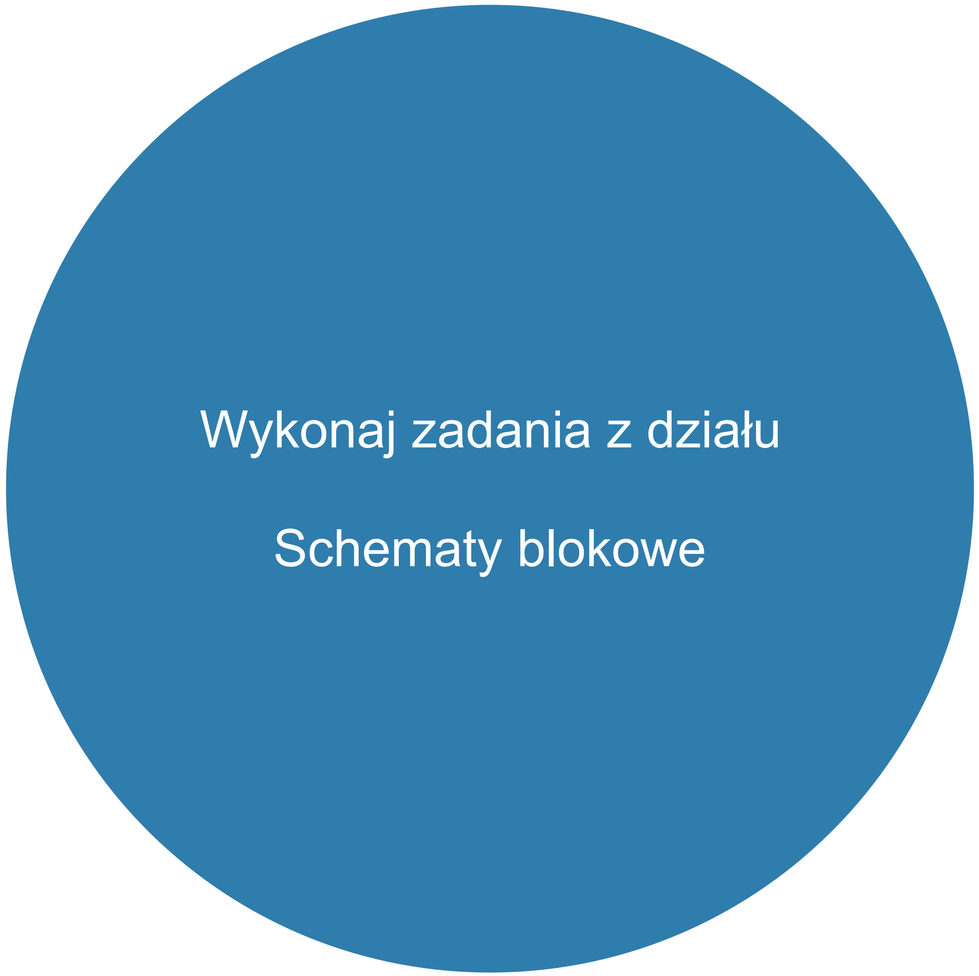
- **Strzałka** – wskazuje jednoznacznie powiązania i ich kierunek.
- **Prostokąt** – zawiera wszystkie operacje z wyjątkiem instrukcji wyboru.
- **Równoległobok** – wejście/wyjście danych.
- **Romb** – wpisujemy wyłącznie instrukcje wyboru.
- **Owal** – oznacza początek bądź koniec schematu.



Schemat blokowy (flowchart)

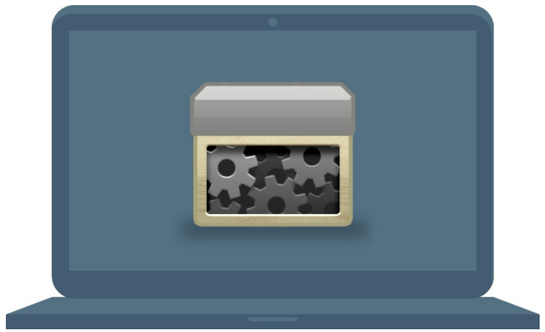
- Algorytm to idea działania programu.
- Przykład implementacji algorytmu może być zapisany w pseudokodzie.
- Pseudokod algorytmu można przełożyć praktycznie na każdy język programowania.

Zadania



Program

Program to wynik kodu źródłowego, który jest tłumaczony (przez inny program, np. kompilator lub interpreter) na kod maszynowy zrozumiały przez komputer i gotowy do uruchomienia.

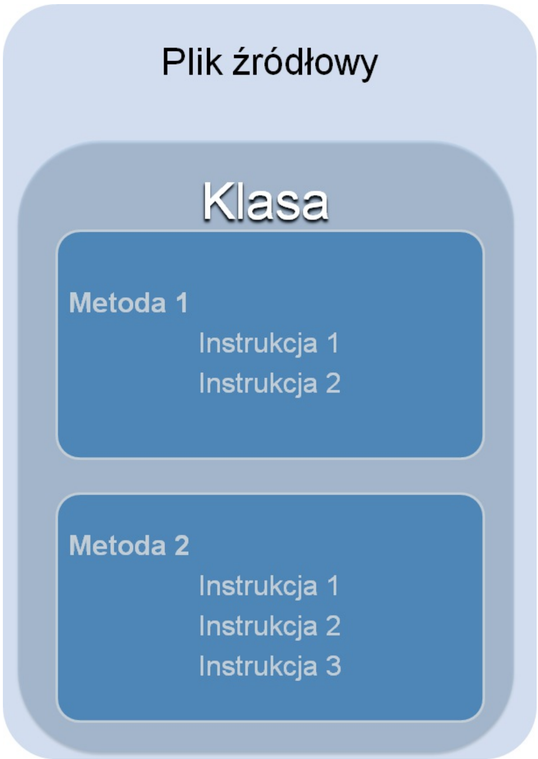


Struktura kodu

Plik źródłowy – czyli ten, w którym będziemy umieszczać nasze programy, zawiera klasę – początkowo możemy ją rozumieć jako element składowy programu.

Nasze pierwsze programy będą się składać z jednej klasy.

Wewnątrz klasy umieszczamy jedną lub wiele metod – są to zestawy instrukcji jakie ma wykonać nasz program.



Pierwszy program

Plik **`MyFirstJavaProgram.java`**:

```java
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **`class, static, void, public,`** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

Pierwszy program

Plik **MyFirstJavaProgram.java**:

```
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **class, static, void, public,** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

MyFirstJavaProgram – to jest nazwa klasy – musi się ona nazywać tak samo jak plik;

Pierwszy program

Plik **MyFirstJavaProgram.java**:

```
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **class, static, void, public,** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

main – nazwa metody;

Pierwszy program

Plik **MyFirstJavaProgram.java**:

```
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **class, static, void, public,** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

MyFirstJavaProgram – nasza metoda **main** przyjmuje tablicę elementów typu String o nazwie **args** – jest to specjalna startowa metoda naszego programu;

Pierwszy program

Plik **MyFirstJavaProgram.java**:

```
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **class, static, void, public,** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

System.out.println – wyświetlenie danych na standardowym wyjściu;

Pierwszy program

Plik **MyFirstJavaProgram.java**:

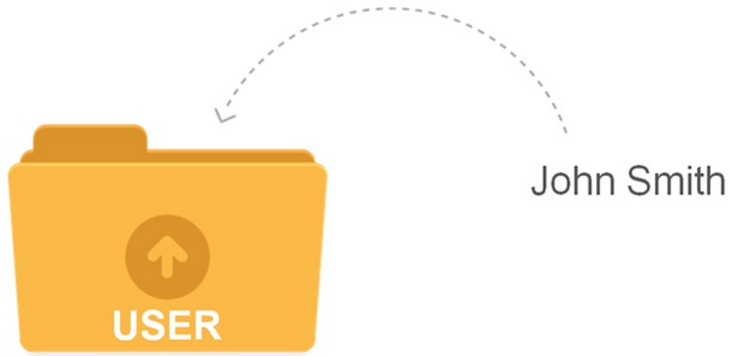
```
public class MyFirstJavaProgram {
    public static void main(String[] args) {
        System.out.println("test");
    }
}
```

Elementy: **class, static, void, public,** zostaną omówione w dalszej części kursu – na razie przyjmijmy, że są to elementy wymagane do uruchomienia programu.

"test" – tekst wyświetlany na standardowym wyjściu – pojawia się w wierszu poleceń systemu;

Zmienne

- Zmienna, w pewnym uproszczeniu, to etykieta odnosząca się do jakiegoś obszaru pamięci.
- Możemy ją zobrazować jako pudełko, które coś zawiera. Takie pudełko ma na sobie nalepkę ze swoją nazwą.
- Chcąc uzyskać dostęp do zawartości pudełka, musimy odwołać się do niego za pomocą jego nazwy.



Porównanie języków

PHP	C++	Java	Co robi?
`print("cześć");`	`printf("cześć");`	`System.out.print("cześć");`	Wpisuje na ekranie słowo `cześć`
`$arr = array(3,2,1);`	`int arr[] = {3,2,1};`	`int[] arr = {3,2,1};`	Tworzy tablicę z danymi: 3,2,1
`sort($arr);`	`std::sort (arr, arr + 3);`	`Arrays.sort(arr);`	Sortuje podaną tablicę z danymi
`if (warunek) { operacja; }`	`if (warunek) { operacja; }`	`if (warunek) { operacja; }`	Wykona daną operację, o ile warunek zostanie spełniony

Instrukcja warunkowa – if... else...

Instrukcja warunkowa **`if`** pozwala na wykonanie kawałka kodu w zależności od warunku postawionego po słowie kluczowym **`if`**:

```
if (instrukcja warunkowa 1) {
    // kod, który się wykona jeśli instrukcja warunkowa 1 jest prawdziwa;
} else if (instrukcja warunkowa 2) {
    // kod, który się wykona jeśli instrukcja warunkowa 2 jest prawdziwa;
} else {
    // kod, który się wykona jeśli instrukcja warunkowa 1 i
    // instrukcja warunkowa 2 są fałszywe;
}
```

Instrukcja warunkowa – if... else...

Przykład w języku naturalnym

Jeśli uzyskasz więcej niż 60 punktów – ocena 5, jeśli więcej niż 50 – ocena 4, jeśli więcej niż 40 – ocena 3, jeśli żadne z powyższych nie zostało spełnione – niestety 2.

Przykład w języku Java

```
int points = 61;
char grade;
if (points >= 60) {
    grade = '5';
} else if (points >= 50) {
    grade = '4';
} else if (points >= 40) {
    grade = '3';
} else {
    grade = '2';
}
System.out.println("Grade = " + grade);
```

Instrukcja warunkowa – switch

Oprócz instrukcji **`if`** mamy również do dyspozycji instrukcję **`switch`**. Używamy jej najczęściej wtedy, kiedy mamy do wyboru więcej opcji niż dwie lub trzy. Zobacz konstrukcję switcha obok:

Przykład w języku Java

```
int dayOfWeek = 2;
String dayOfWeekString;
switch (dayOfWeek) {
case 1:  dayOfWeekString = "Monday";
    break;
case 2:  dayOfWeekString = "Tuesday";
    break;
case 3:  dayOfWeekString = "Wednesday";
    break;
default: dayOfWeekString = "Invalid";
    break;
}
System.out.println(dayOfWeekString);
```

Instrukcja warunkowa – switch

Oprócz instrukcji **`if`** mamy również do dyspozycji instrukcję **`switch`**. Używamy jej najczęściej wtedy, kiedy mamy do wyboru więcej opcji niż dwie lub trzy. Zobacz konstrukcję switcha obok:

Po każdym warunku **`case`** pamiętaj o słowie kluczowym **`break`**, za pomocą którego możemy wyjść z instrukcji **`switch`**. Nie jest obowiązkowe, ale w przypadku jego braku każda instrukcja zostanie wykonana.

Przykład w języku Java

```
int dayOfWeek = 2;
String dayOfWeekString;
switch (dayOfWeek) {
case 1:  dayOfWeekString = "Monday";
    break;
case 2:  dayOfWeekString = "Tuesday";
    break;
case 3:  dayOfWeekString = "Wednesday";
    break;
default: dayOfWeekString = "Invalid";
    break;
}
System.out.println(dayOfWeekString);
```

Instrukcja warunkowa – switch

Oprócz instrukcji **`if`** mamy również do dyspozycji instrukcję **`switch`**. Używamy jej najczęściej wtedy, kiedy mamy do wyboru więcej opcji niż dwie lub trzy. Zobacz konstrukcję switcha obok:

Po każdym warunku **`case`** pamiętaj o słowie kluczowym **`break`**, za pomocą którego możemy wyjść z instrukcji **`switch`**. Nie jest obowiązkowe, ale w przypadku jego braku każda instrukcja zostanie wykonana.

Słówko kluczowe **`default`** zostaje wykonane wtedy, gdy żaden warunek nie pasuje do **`case.`**

Przykład w języku Java

```
int dayOfWeek = 2;
String dayOfWeekString;
switch (dayOfWeek) {
case 1:  dayOfWeekString = "Monday";
    break;
case 2:  dayOfWeekString = "Tuesday";
    break;
case 3:  dayOfWeekString = "Wednesday";
    break;
default: dayOfWeekString = "Invalid";
    break;
}
System.out.println(dayOfWeekString);
```

Pętla for

```
for(inicjalizacja zmiennych; sprawdzenie warunku; modyfikacja zmiennych) {
      //instrukcje do wykonania w pętli
}
```

- Przed rozpoczęciem wykonywania kodu pętli wartościowane jest **wyrażenie_inicjujące.**
- Następnie sprawdzana jest wartość **wyrażenia_warunkowego.** Jeżeli jest **`true`**, pętla może być kontynuowana – wykonywane są instrukcje w ciele pętli. W przeciwnym razie (**`false`**) wykonanie pętli zostaje zakończone.
- Kolejnym krokiem jest wartościowanie **wyrażenia_iteracyjnego** (najczęściej modyfikacja licznika pętli) i znowu obliczane jest **wyrażenie_warunkowe** decydujące, czy pętla będzie kontynuowała swoje działanie.
- **Wyrażenie_inicjujące** wartościowane jest tylko raz, przed wykonaniem pętli.

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

`int i = 0`

Inicjalizacja – zmiennej `i` przypisujemy wartość 0.

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

`int i = 0`

Inicjalizacja – zmiennej **`i`** przypisujemy wartość 0.

`i <= 5`

Warunek – sprawdzamy czy wartość **`i`** w danej iteracji jest mniejsza bądź równa 5.

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

`int i = 0`
Inicjalizacja – zmiennej **`i`** przypisujemy wartość 0.

`i <= 5`
Warunek – sprawdzamy czy wartość **`i`** w danej iteracji jest mniejsza bądź równa 5.

`i++`
Modyfikacja – zwiększamy wartość zmiennej **`i`** o jeden w każdej iteracji.

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

`int i = 0`
Inicjalizacja – zmiennej **`i`** przypisujemy wartość 0.

`i <= 5`
Warunek – sprawdzamy czy wartość **`i`** w danej iteracji jest mniejsza bądź równa 5.

`i++`
Modyfikacja – zwiększamy wartość zmiennej **`i`** o jeden w każdej iteracji.

Zwyczajowo zmienne które sterują pętlą nazywamy **`i, j`** itd.

Pętla for – przykład

```
for (int i = 0; i <= 5; i++) {
    System.out.print(i + " ");
}
```

`int i = 0`
Inicjalizacja – zmiennej **`i`** przypisujemy wartość 0.

`i <= 5`
Warunek – sprawdzamy czy wartość **`i`** w danej iteracji jest mniejsza bądź równa 5.

`i++`
Modyfikacja – zwiększamy wartość zmiennej **`i`** o jeden w każdej iteracji.

Zwyczajowo zmienne które sterują pętlą nazywamy **`i, j`** itd.

Wynik: **`0 1 2 3 4 5`**

Pętla while

Pętlę **`while`** wykorzystujemy, jeżeli nie wiemy, ile razy będziemy wykonywać jakieś instrukcje.

Pętla jest wykonywana do momentu, gdy podany w nawiasach warunek jest prawdziwy.

```
int i = 0;
while (i < 5) {
    System.out.println("i = " + i);
    i++;
}
```

Pętla while

Pętlę **`while`** wykorzystujemy, jeżeli nie wiemy, ile razy będziemy wykonywać jakieś instrukcje.

Pętla jest wykonywana do momentu, gdy podany w nawiasach warunek jest prawdziwy.

```
int i = 0;
while (i < 5) {
    System.out.println("i = " + i);
    i++;
}
```

Dopóki **`i`** jest mniejsze od 5, wykonuj pętlę.

Wyświetlanie napisów

Wyświetlanie w konsoli napisu „**test**” zakończone przejściem do nowej linii:

```
System.out.println("test");
```

Wyświetlanie w konsoli napisu „**test**” bez przejścia do nowej linii:

```
System.out.print("test");
```

Zapamiętaj te instrukcje – w początkowej fazie nauki będziemy z nich często korzystali.

Wyświetlanie napisów

Wyświetlanie w konsoli napisu „**test**” zakończone przejściem do nowej linii:

```
System.out.println("test");
```

Wyświetlanie w konsoli napisu „**test**” bez przejścia do nowej linii:

```
System.out.print("test");
```

Zwróć uwagę, że napis przekazany do wyświetlenia jest umieszczony w cudzysłowie ("").

Zapamiętaj te instrukcje – w początkowej fazie nauki będziemy z nich często korzystali.

Wyświetlanie napisów

Wyświetlanie w konsoli napisu „**test**” zakończone przejściem do nowej linii:

```
System.out.println("test");
```

Wyświetlanie w konsoli napisu „**test**” bez przejścia do nowej linii:

```
System.out.print("test");
```

Zwróć uwagę, że napis przekazany do wyświetlenia jest umieszczony w cudzysłowie ("").

Każda linia instrukcji musi się kończyć znakiem średnika (;).

Zapamiętaj te instrukcje – w początkowej fazie nauki będziemy z nich często korzystali.

Kompilacja i uruchomienie

Uruchomić program możemy używając do tego celu konsoli.

Aby wykonać powyższy kod, należy go najpierw skompilować za pomocą narzędzia **javac**.

Przechodzimy w strukturze katalogów do miejsca, w którym znajduje się plik źródłowy, następnie wykonujemy polecenie:

```
javac MyFirstJavaProgram.java
```

W wyniku tej operacji dostaniemy skompilowany plik **`MyFirstJavaProgram.class`**, który możemy uruchomić za pomocą narzędzia **java**:

```
java MyFirstJavaProgram
```

Pamiętaj, że wielkość liter ma znaczenie.

Uwaga: uruchamiając program, nie podajemy żadnego rozszerzenia.

IntelliJ IDEA

Developerzy w swojej pracy korzystają z tzw. **IDE** (**Integrated Development Environment**) – jest to zintegrowane środowisko programistyczne służące do tworzenia programów w Javie i nie tylko.

W języku potocznym często posługujemy się skrótem **IDE** – jako programu do pisania programów.

Podczas zajęć będziemy korzystać z IDE **IntelliJ IDEA**.

IntelliJ posiada bardzo bogatą bazę rozszerzeń, która jest już wbudowana w platformę, dzięki czemu oferuje wsparcie dla wielu frameworków, bez konieczności instalowania dodatkowych wtyczek.

Popularne IDE

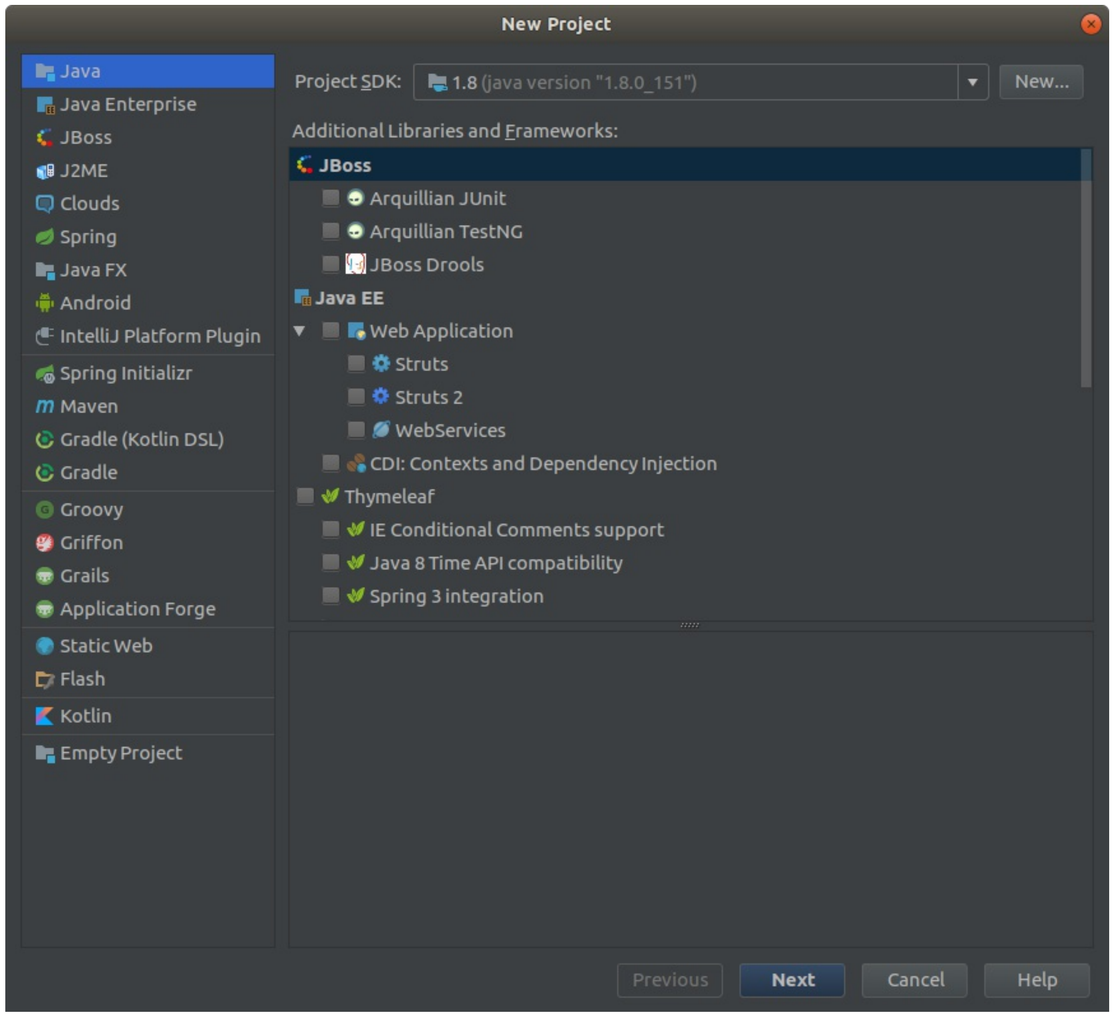
Oprócz IntelliJ IDEA najbardziej znane środowiska to:

Eclipse – najpopularniejsze darmowe IDE, posiada bardzo bogatą bazę rozszerzeń do doinstalowania.

NetBeans – darmowe IDE ze wsparciem dla Swing GUI Builder – posiada wizualny edytor do biblioteki graficznej SWING.

STS – darmowe IDE oparte na Eclipse z wtyczką wspierającą programowanie z użyciem frameworka Spring.

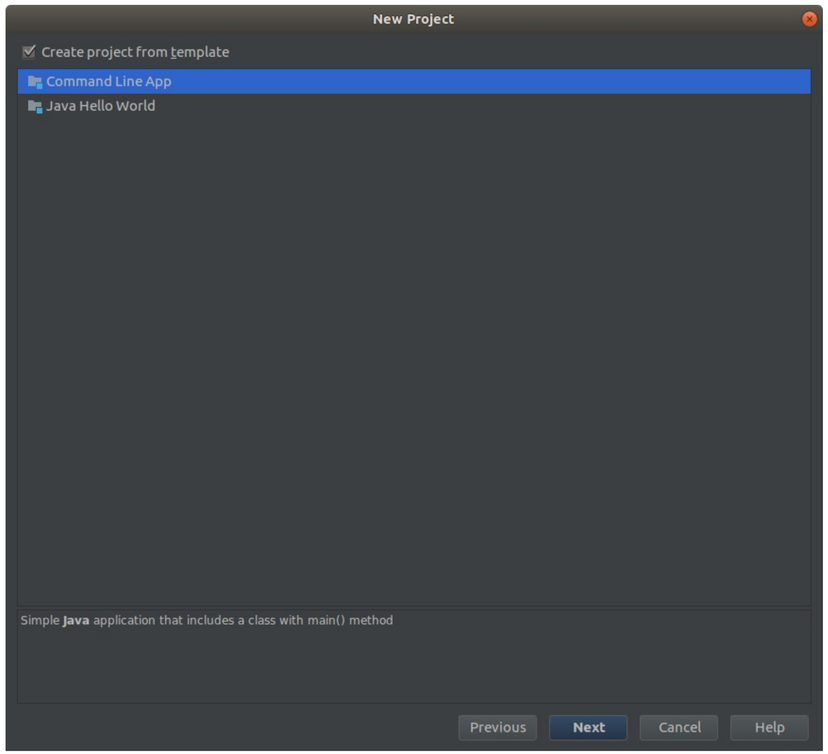
IntelliJ – nowy projekt



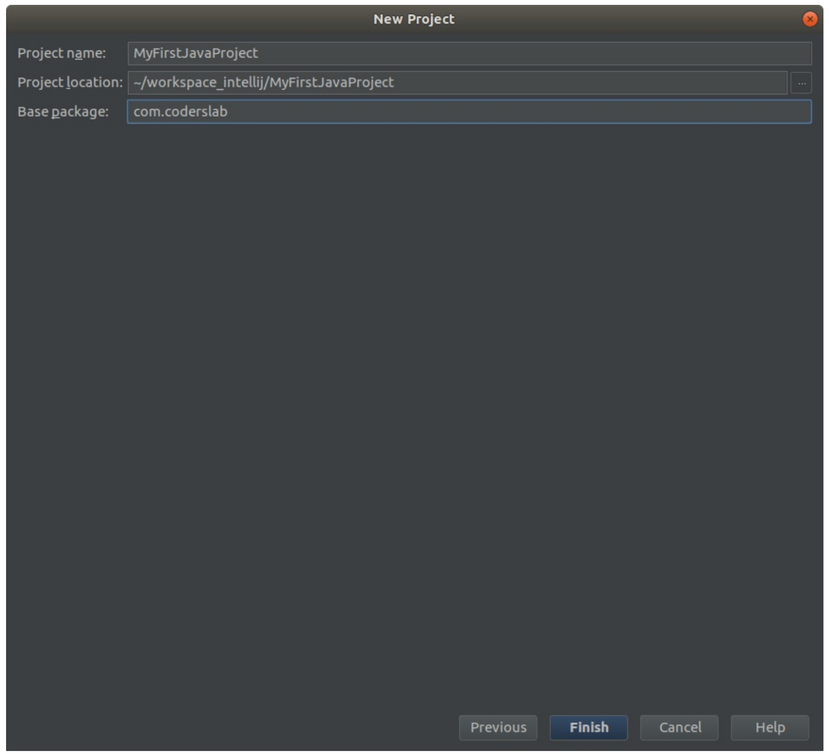
Utworzenie nowego projektu

- Z paska narzędzi otwórz menu **File**
- Wybierz **New –> Project...**
- Pojawi się okno jak po lewej stronie, przejdź dalej, klikając przycisk **Next**.

IntelliJ – nowy projekt



Wybierz opcję **Create project from template** i wciśnij **Next**.



Wpisz nazwę projektu i wybierz jego lokalizację, a następnie zatwierdź utworzenie nowego projektu klikając **Finish**.

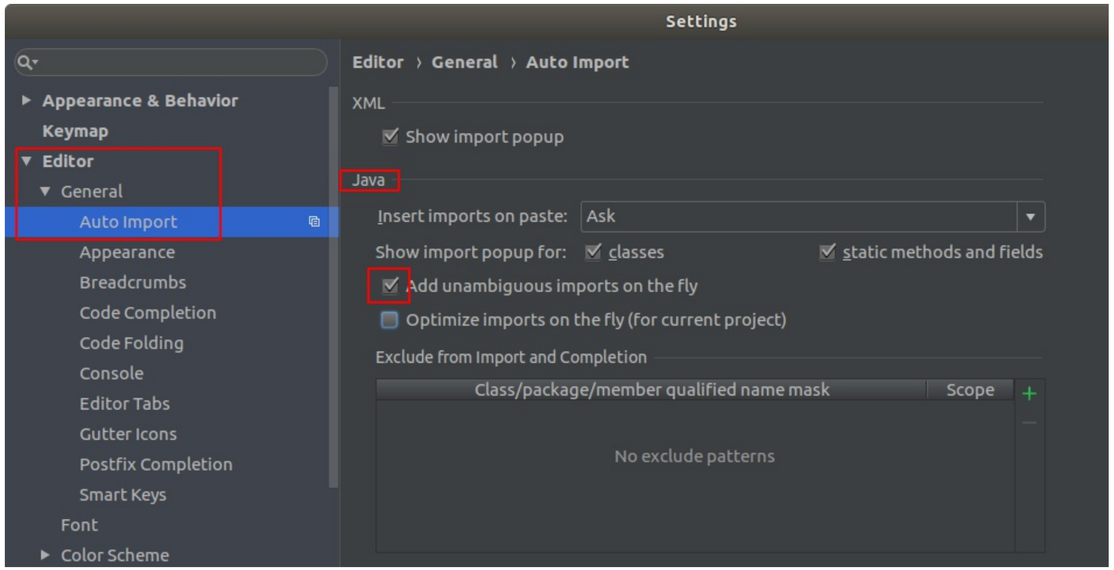
IntelliJ – ustawienia projektu

Dużym usprawnieniem pracy będzie ustawienie w IDE automatycznego dodawania importów – wtedy nie musimy ich ręcznie dodawać do klasy. Import nie zostanie dodany, jeśli IntelliJ nie będzie w stanie jednoznacznie stwierdzić, o który chodzi. Wtedy podpowiada listę możliwych bibliotek do zaimportowania, z których wybieramy właściwą. Automatyzację dodawania importów włączamy wybierając:

`File –> Settings... –> Editor –> General –> Auto Import`

Tam zaznaczamy opcję:

`Add unambiguous imports on the fly`



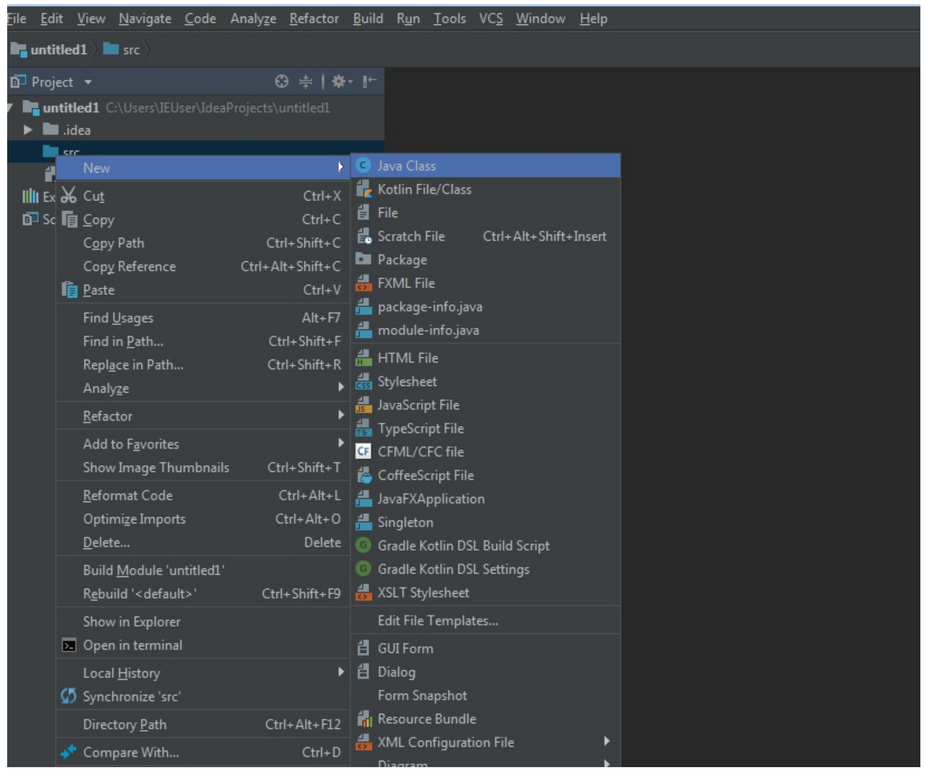
Tworzenie klas – IntelliJ

Aby utworzyć nową klasę za pomocą **IntelliJ** należy wybrać z menu górnego:

`File –> New –> Java Class`

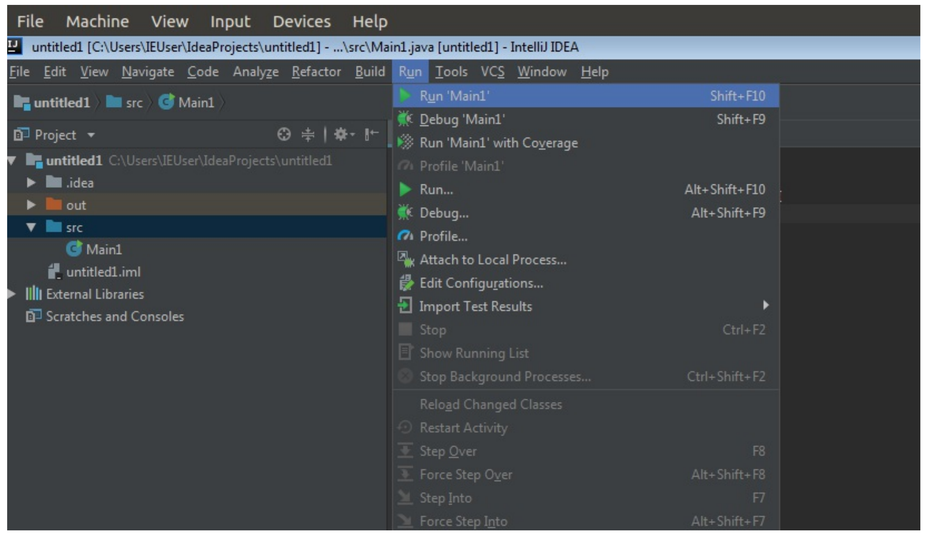
Lub prawym klawiszem myszy na drzewie projektu a następnie:

`New –> Java Class`



Kompilacja i uruchomienie – IntelliJ

Aby uruchomić program z poziomu IntelliJ wystarczy wybrać z menu górnego zakładkę **Run**, a następnie polecenie **Run** na samej górze listy rozwijanej:



Dobrym i szybkim sposobem jest korzystanie ze skrótu klawiszowego: **Shift + F10**

Java – obiektowość

- Java to język obiektowy.
- Oprócz kilku typów prostych omówionych w materiałach przygotowawczych, wszystko jest obiektem.
- Obiekt to element, który posiada cechy – nazywane atrybutami.
- Obiekt posiada również operacje – nazywane metodami.

- Program w Javie musi zawierać się w klasie.
- Klasa to podstawowy element składowy aplikacji.
- W klasach umieszczamy atrybuty (cechy obiektów) oraz metody (fragmenty kodu zajmujące się wykonywaniem pewnych zadań).
- Nasze pierwsze programy będą zbudowane z jednej klasy oraz z jednej metody: **`main`**.
- Wiele klas można połączyć w jeden pakiet.

Są to tymczasowe definicje – będziemy je rozszerzać w dalszych częściach kursu.

Klasy i obiekty

Klasy

- Klasa w języku Java to zbiór ściśle powiązanych ze sobą metod i zmiennych. Klasy umieszcza się w osobnych plikach.
- Przykładem może być klasa reprezentująca użytkownika, zawierająca metody:
  - dodającą użytkownika,
  - logującą użytkownika,
  - zmieniającą hasło użytkownika,
  - wysyłająca mail do użytkownika itd.

Metody

- Metody to wydzielone i nazwane bloki kodu w naszej klasie.

Własności

- Własności w klasie to swego rodzaju zmienne, które są związane tylko i wyłącznie z daną klasą.

Klasy i obiekty

Tworzenie obiektu klasy

- Gdy już posiadamy zdefiniowaną klasę, czyli strukturę metod i zmiennych, to aby z niej skorzystać, musimy utworzyć jej obiekt, czyli pojedynczy egzemplarz naszej klasy.
- Używamy do tego słowa kluczowego **new**, po którym podajemy nazwę klasy, a całe wywołanie przypisujemy do zmiennej.

```
User user = new User();
```

User – określenie typu,

user – zmienna,

new User() – nowy obiekt typu User (ze słowem kluczowym **new**).

Klasy i obiekty

Wywołanie metody

- Wywołanie metody na obiekcie, to wykonanie operacji dotyczącej naszego obiektu, dlatego też metodę wywołujemy na zmiennej z obiektem klasy.
- Metodę wywołujemy dodając kropkę (`.`) po nazwie zmiennej obiektu, a następnie podając nazwę metody i ewentualne parametry.
- Jeśli metoda coś zwraca, przypisujemy jej wywołanie do zmiennej.

Przykład

```
user.setLogin("login");
```

`user` – wywołujemy na zmiennej obiektu,

`setLogin` – po **`"."`** podajemy nazwę metody i ewentualne argumenty.

Przykład innej metody wywołanej na tym samy obiekcie:

```
user.login();
```

Klasy i obiekty

Wywołanie metody

- Skąd wiemy jak nazywa się metoda, którą chcemy wywołać? Wynika to z definicji klasy lub dokumentacji biblioteki.
- Metody mogą przyjmować argumenty, ale nie muszą.

Własności

- Załóżmy, że nasz użytkownik posiada cechy, które go charakteryzują np. wiek, płeć, miasto zamieszkania – do ich przypisania użyjemy własności.
- Przypisanie własności to dodanie kropki (`.`) do nazwy zmiennej obiektu, po której następuje nazwa własności i standardowe przypisanie wartości.

Klasy i obiekty

Przypisanie wartości:

```
user.age = 25;
user.city = "Warsaw";
user.sex = "male";
```

Wywołujemy na zmiennej **`user`**.

Nazwę własności podajemy po kropce (`.`)

Pobranie wartości z własności odbywa się w analogiczny sposób:

```
int userAge = user.age;
String userCity = user.city;
String userSex = user.sex;
```

Referencje

Deklaracja zmiennej nie powoduje utworzenia obiektu – otrzymujemy tylko referencję:

```
User user;
```

Dopiero używając słowa kluczowego **new** oraz poniższej konstrukcji:

```
User user = new User();
```

przypisujemy do referencji **user** nowy obiekt klasy **User**.

Wartościami zmiennych typu obiektowego są albo referencje (odnośniki) do obiektów, albo **null**.

Referencja może istnieć samodzielnie.

Po utworzeniu zmiennej obiektowej pamiętaj, że tworzysz odwołanie do obiektu. Dopóki nie nadasz tej zmiennej wartości, ma ona wartość **null**.

(**null** reprezentuje pustą wartość – czyli brak obiektu przypisanego do zmiennej)

Uwaga! null nie oznacza 0 (zero).

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Tworzymy zmienne **`variable1`** i **`variable2`**, dodatkowo przypisujemy od razu wartość zmiennej **`variable2`**.

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Tworzymy zmienne **`variable1`** i **`variable2`**, dodatkowo przypisujemy od razu wartość zmiennej **`variable2`**.

Kopiujemy wartość zmiennej **`variable2`** do **`variable1`**.

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Tworzymy zmienne **`variable1`** i **`variable2`**, dodatkowo przypisujemy od razu wartość zmiennej **`variable2`**.

Kopiujemy wartość zmiennej **`variable2`** do **`variable1`**.

Późniejsza modyfikacja **`variable2`** nie wpływa na wartość **`variable1`**.

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Tworzymy zmienne **`variable1`** i **`variable2`**, dodatkowo przypisujemy od razu wartość zmiennej **`variable2`**.

Kopiujemy wartość zmiennej **`variable2`** do **`variable1`**.

Później modyfikacja **`variable2`** nie wpływa na wartość **`variable1`**.

Zwróci 10.

Przypisanie

Przypisanie – typ prosty

```
int variable1, variable2 = 10;
variable1 = variable2;
variable2 = 1;
System.out.println(variable1);
System.out.println(variable2);
```

Tworzymy zmienne **variable1** i **variable2**, dodatkowo przypisujemy od razu wartość zmiennej **variable2**.

Kopiujemy wartość zmiennej **variable2** do **variable1**.

Późniejsza modyfikacja **variable2** nie wpływa na wartość **variable1**.

Zwróci 10.

Zwróci 1.

Przypisanie

Przypisanie – typy obiektowe

```
User user1 = new User();
user1.age = 100;
User user2 = user1;
user2.age = 25;
System.out.println(user1.age);
```

Przypisanie

Przypisanie – typy obiektowe

```
User user1 = new User();
user1.age = 100;
User user2 = user1;
user2.age = 25;
System.out.println(user1.age);
```

Skopiowanie do **`user2`** referencji do obiektu, wskazywanego przez **`user1`**. Obie zmienne wskazują teraz na ten sam obiekt.

Przypisanie

Przypisanie – typy obiektowe

```
User user1 = new User();
user1.age = 100;
User user2 = user1;
user2.age = 25;
System.out.println(user1.age);
```

Skopiowanie do **`user2`** referencji do obiektu, wskazywanego przez **`user1`**. Obie zmienne wskazują teraz na ten sam obiekt.

Modyfikacja obiektu wskazywanego przez **`user2`** powoduje modyfikacje obiektu wskazywanego przez **`user1`**. Modyfikacja **`user2.age`** wpływa na wartość **`user1.age`**.

Przypisanie

Przypisanie – typy obiektowe

```
User user1 = new User();
user1.age = 100;
User user2 = user1;
user2.age = 25;
System.out.println(user1.age);
```

Skopiowanie do **`user2`** referencji do obiektu, wskazywanego przez **`user1`**. Obie zmienne wskazują teraz na ten sam obiekt.

Modyfikacja obiektu wskazywanego przez **`user2`** powoduje modyfikacje obiektu wskazywanego przez **`user1`**. Modyfikacja **`user2.age`** wpływa na wartość **`user1.age`**.

zwróci 25

Parametry przy uruchamianiu

Zdarzyć się może, że będziemy chcieli uruchomić nasz program podając dla niego pewne parametry.

Uruchomienie programu z poziomu konsoli:

```
java MyProgram arg1 arg2 arg3
```

Dostęp do tych parametrów mamy poprzez argument metody **`main`** naszego programu.

```
public static void main(String[] args) {
    for (String param : args) {
        System.out.println(param);
    }
}
```

Parametry przy uruchamianiu

Zdarzyć się może, że będziemy chcieli uruchomić nasz program podając dla niego pewne parametry.

Uruchomienie programu z poziomu konsoli:

```
java MyProgram arg1 arg2 arg3
```

Dostęp do tych parametrów mamy poprzez argument metody **main** naszego programu.

```
public static void main(String[] args) {
    for (String param : args) {
        System.out.println(param);
    }
}
```

args – tablica zawierająca parametry podane podczas uruchamiania.

Słowa kluczowe

Jak w każdym języku programowania, także w języku **Java** występuje zestaw słów zarezerwowanych dla języka, są to np.:

`boolean, break, byte, case, catch, do, double, else, enum, extends, if, implements, import, instanceof, int, private, protected, public, return, short, this, throw, throws, transient, try` itd.

Pełen zestaw znajdziemy pod adresem: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Stałe

Stała to element naszego programu, któremu podczas działania programu nie może zostać przypisana inna wartość.

Stałe tworzymy przy użyciu słów kluczowych **`static final`**:

```
static final double PI = 3.146;
```

Typ określamy tak, jak w przypadku innych zmiennych.

Metody

Metody

W programowaniu często spotykamy się z sytuacją, że jakąś funkcjonalność chcemy wykorzystać w kilku miejscach naszego programu. Wtedy, zamiast kopiować kod w każde z tych miejsc, powinniśmy skorzystać z metod.

- Metoda to wydzielony fragment kodu o zdefiniowanej nazwie, który może być używany w różnych momentach i miejscach programu.
- Metodę można opisać jako rozwiązanie jakiegoś ogólnego problemu – przy tym dokładne informacje (zmienne), potrzebne do rozwiązania, będą podane dopiero przy użyciu tej metody.

Metody

- Metodę definiuje się raz, a później wywołuje się ją dowolną liczbę razy, w zależności od potrzeb.
- Sygnatura metody zawiera: typ zwracany, nazwę metody i typy argumentów. Metoda musi posiadać unikatową sygnaturę.
- Pisanie metod zgodnie z dobrymi praktykami daje nam możliwość ponownego ich użycia, a także znacznie poprawia przejrzystość kodu.

- Zastosowanie metod powoduje, że w przypadku konieczności zmiany kodu, zmieniamy go tylko w metodzie, a nie w każdym miejscu gdzie zamiast prostego wywołania metody, skopiowalibyśmy nasz kod.
- **Metody definiujemy wewnątrz naszej klasy.**

Metody – definiowanie

- Nazwa metody może się składać z małych lub dużych liter, cyfr i znaków podkreślenia. Musi się zaczynać albo od litery, albo od znaku podkreślenia.
- Przed nazwą określamy typ zwracanej wartości (np.: **`int`**, **`String`**, **`boolean`** itd.) lub jej brak (za pomocą słowa kluczowego **`void`**).

```
int methodName(int param1, int param2,
           /* ... */, int paramN)
{
    instruction1;
    instruction2;
    ...
    instructionN;
    return result;
}
```

Instrukcja **`return`** zwraca wartość (wynik działania) metody i kończy jej działanie. Wszelkie instrukcje umieszczone za instrukcją **`return`** nie zostaną wykonane.

Metody – definiowanie

- Po nazwie metody następuje lista parametrów, które metoda ma przyjmować. Parametry oddzielamy od siebie przecinkiem.
- Listę parametrów obejmujemy nawiasami okrągłymi:
  `method1(int param1, String param2, boolean param3);`
- Jeżeli metoda nie przyjmuje żadnych parametrów, umieszczamy za nazwą metody puste nawiasy:
  `method2();`

Dwa typy metod

Metody w programowaniu możemy podzielić na dwa typy:

Metody, które nie zwracają wartości:

Są to metody, w których interesuje nas efekt uboczny wywołania takiej metody. Definiujemy je używając słowa kluczowego **`void`**. Może być to np.:

- wyświetlenie wiadomości na ekranie,
- wysłanie maila przez serwer,
- zapisanie informacji do bazy danych.

Metody, które zwracają wartość:

Są to metody, w których interesuje nas zwracana wartość. Używają one słowa kluczowego **`return`** i mają określony typ zwracanej wartości. Mogą być to np.:

- najróżniejsze obliczenia matematyczne (potęgowanie, pierwiastkowanie itd.),
- wczytywanie informacji z bazy danych,
- wyszukiwanie danych w zbiorze (tablicy).

Dwa typy metod

Metody, które nie zwracają wartości:

```
public static void main(String[] args) {
    String name = "User";
    sayHello(name);
}
static void sayHello(String userName) {
    System.out.println("Hello " + userName);
}
```

Dwa typy metod

Metody, które nie zwracają wartości:

```
public static void main(String[] args) {
    String name = "User";
    sayHello(name);
}
static void sayHello(String userName) {
    System.out.println("Hello " + userName);
}
```

Metoda **nie zwraca** wartości więc po prostu ją wywołujemy.

Dwa typy metod

Metody, które nie zwracają wartości:

```
public static void main(String[] args) {
    String name = "User";
    sayHello(name);
}
static void sayHello(String userName) {
    System.out.println("Hello " + userName);
}
```

Metoda **nie zwraca** wartości więc po prostu ją wywołujemy.

Metoda używa słowa kluczowego **`void`** czyli **nie zwraca** żadnej wartości.

Dwa typy metod

Metody które zwracają wartość:

```
int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

Dwa typy metod

Metody które zwracają wartość:

```
int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

Metoda używa słowa kluczowego **`return`** i ma określony typ zwracanej wartości (**`int`**).

Dwa typy metod

Metody które zwracają wartość:

```
int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

Metoda używa słowa kluczowego **`return`** i ma określony typ zwracanej wartości (**`int`**).

Jako że interesuje nas wynik tej metody, to wartość przez nią **zwracaną** zapisujemy do zmiennej – **`int sum`**. Zmiennej tej możemy potem użyć w dalszej części naszego kodu.

Metody – definiowanie

Definiując metodę, przed jej nazwą – oprócz typu – możemy określić modyfikatory dostępu np. **`public`**, **`private`** itd.

Elementy te omówimy dokładnie w późniejszych działach. W początkowej fazie nauki możemy całkowicie pominąć ich wpisywanie oprócz metody **`main`**.

Przy próbie wywołania metody, która deklaruje inny typ niż rzeczywiście zwracany np.:

```
static int methodName() {
      return "String";
}
```

otrzymamy błąd kompilacji:

```
Exception in thread "main" java.lang.Error:
        Unresolved compilation problem:
        Type mismatch: cannot convert from String to int
```

Metody – definiowanie

```
void method1() {
    System.out.println("Nic nie zwraca.");
}
int method2() {
    return 321;
}
String method3() {
    return "Coderslab";
}
```

Metody – definiowanie

```
void method1() {
    System.out.println("Nic nie zwraca.");
}
int method2() {
    return 321;
}
String method3() {
    return "Coderslab";
}
```

Za pomocą słowa kluczowego **`void`** definiujemy metodę, która nic nie zwraca.

Metody – definiowanie

```
void method1() {
    System.out.println("Nic nie zwraca.");
}
int method2() {
    return 321;
}
String method3() {
    return "Coderslab";
}
```

Za pomocą słowa kluczowego **`void`** definiujemy metodę, która nic nie zwraca.

Określamy typ zwracanej wartości – **`int`**, **`String`**.

Metody – definiowanie

```
void method1() {
    System.out.println("Nic nie zwraca.");
}
int method2() {
    return 321;
}
String method3() {
    return "Coderslab";
}
```

Za pomocą słowa kluczowego **`void`** definiujemy metodę, która nic nie zwraca.

Określamy typ zwracanej wartości – **`int`**, **`String`**.

Za pomocą słowa kluczowego **`return`** zwracamy wartość.

Metody – static

- Metodę możemy określić jako statyczną (nazywaną również klasową) dodając przed jej nazwą słowo kluczowe **`static`**.
- Metody statyczne można wywołać bez tworzenia obiektu.
- Przykładem takiej metody jest metoda **`join`** klasy String, którą będziemy omawiać na zajęciach.

W dokumentacji metoda schematycznie przedstawia się następująco:

```
public static String join(CharSequence delimiter, CharSequence... elements)
```

Przykład

```
String joined = String.join("-", "Coders", "Lab");  //wynik: Coders-Lab
```

Zwróćmy uwagę, że nie definiujemy nowego obiektu przy użyciu słowa kluczowego **`new`**.

Metody – static

- Metodę możemy określić jako statyczną (nazywaną również klasową) dodając przed jej nazwą słowo kluczowe **`static`**.
- Metody statyczne można wywołać bez tworzenia obiektu.
- Przykładem takiej metody jest metoda **`join`** klasy String, którą będziemy omawiać na zajęciach.

W dokumentacji metoda schematycznie przedstawia się następująco:

```
public static String join(CharSequence delimiter, CharSequence... elements)
```

Za pomocą słowa kluczowego **`static`** określamy, że metoda jest statyczna.

Przykład

```
String joined = String.join("-", "Coders", "Lab");  //wynik: Coders-Lab
```

Zwróćmy uwagę, że nie definiujemy nowego obiektu przy użyciu słowa kluczowego **`new`**.

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

Wywołujemy metodę.

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

Definiujemy metodę.

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

Wywołujemy metodę.

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Definiujemy metodę.

Metody static – wywoływanie

Statyczną metodę możemy wywołać (czyli uruchomić) w dowolnym miejscu metody **main** naszego programu lub każdej innej **metody statycznej**.

```
public static void main(String[] args){
    method1();
}
static void method1() {
    System.out.println("Metoda nic" +
                       "nie zwraca.");
}
static String method3() {
    method1();
    return "Coderslab";
}
```

```
static int method2() {
    return 321;
}
public static void main(String[] args){
    int variable = method2();
}
```

Tworzymy zmienną **variable** i przypisujemy jej wynik wywołania metody **method2()**.

Metody – parametry i argumenty

Do metody możemy przekazywać parametry – ich liczba może być dowolna. Wówczas, podczas uruchomienia programu, musimy podać tyle argumentów, ile parametrów zostało zadeklarowanych. Argumenty to zmienne, potrzebne do prawidłowego działania metody.

O parametrach musicie myśleć jak o zmiennych, na których będzie operować wasza metoda. Pracuje się na nich jak na każdej innej zmiennej, z tym wyjątkiem, że do chwili użycia metody nie znamy ich dokładnej wartości.

```
static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

Metody – parametry i argumenty

Do metody możemy przekazywać parametry – ich liczba może być dowolna. Wówczas, podczas uruchomienia programu, musimy podać tyle argumentów, ile parametrów zostało zadeklarowanych. Argumenty to zmienne, potrzebne do prawidłowego działania metody.

O parametrach musicie myśleć jak o zmiennych, na których będzie operować wasza metoda. Pracuje się na nich jak na każdej innej zmiennej, z tym wyjątkiem, że do chwili użycia metody nie znamy ich dokładnej wartości.

```
static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

Deklarując naszą metodę **sum**, definiujemy dla niej dwa parametry: **a** i **b**;

Metody – parametry i argumenty

Do metody możemy przekazywać parametry – ich liczba może być dowolna. Wówczas, podczas uruchomienia programu, musimy podać tyle argumentów, ile parametrów zostało zadeklarowanych. Argumenty to zmienne, potrzebne do prawidłowego działania metody.

O parametrach musicie myśleć jak o zmiennych, na których będzie operować wasza metoda. Pracuje się na nich jak na każdej innej zmiennej, z tym wyjątkiem, że do chwili użycia metody nie znamy ich dokładnej wartości.

```
static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

a + **b** – wewnątrz metody korzystamy ze zmiennych, które do niej przekazaliśmy;

Metody – parametry i argumenty

Do metody możemy przekazywać parametry – ich liczba może być dowolna. Wówczas, podczas uruchomienia programu, musimy podać tyle argumentów, ile parametrów zostało zadeklarowanych. Argumenty to zmienne, potrzebne do prawidłowego działania metody.

O parametrach musicie myśleć jak o zmiennych, na których będzie operować wasza metoda. Pracuje się na nich jak na każdej innej zmiennej, z tym wyjątkiem, że do chwili użycia metody nie znamy ich dokładnej wartości.

```
static int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
int a = 3, b = 2;
int sum = sum(a, b);
```

a, **b** – wywołując metodę podajemy dwa argumenty: **a** i **b**.

Zadania

Tablice

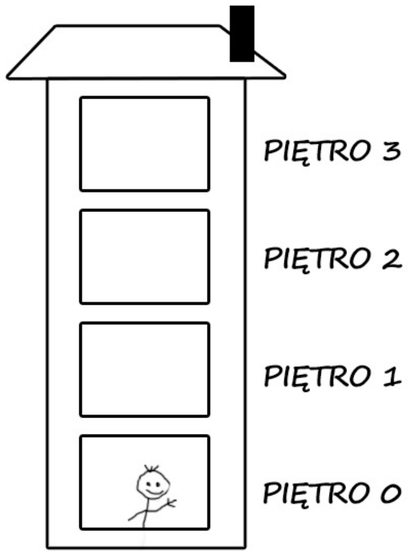
Tablice

Czym są tablice?

Tablice to struktury danych przechowujące zbiór danych jednego typu, do których możemy się odwoływać poprzez jedną wspólną nazwę tablicy. Każdy element tablicy jest zapisany pod unikalnym indeksem.

Każdy indeks (klucz) odwołuje się do przypisanej mu wartości.

Tablicę można porównać do kamienicy, w której na każdym piętrze mieszka jeden lokator. Jak dowiedzieć się, kto mieszka w tej kamienicy na drugim piętrze?



Tablice

Tablicę deklarujemy jak zwykłą zmienną, ale wstawiamy do niej specjalną konstrukcję **[]** (nawiasy kwadratowe), wg schematu: **typ[] nazwa;**

Np.:

- **int[] arrayOfInt;**
- **long[] arrayOfLongs;**
- **float[] arrayOfFloats;**
- **boolean[] arrayOfBooleans;**
- **String[] arrayOfStrings;**

Możemy również umieścić symbole **[]** za nazwą:

```
int arrayOfInt[];
```

Przykład tworzenia tablicy pięcioelementowej typu **int**:

```
int[] myNumbers = new int[5];
```

W tym momencie mamy pięcioelementową tablicę, której elementy są zerami.

Wartości domyślne będą różne dla poszczególnych typów – szczegóły znajdziemy na kolejnych slajdach.

Tablice i pętle

Jeżeli chcielibyśmy w prosty sposób wypisać wszystkie elementy tablicy w konsoli, możemy użyć pętli. Zobacz przykład poniżej:

```
int numbers[] = { 120, 3, 45 };
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + ", ");
}
```

Zauważ, że zmienną **`i`** ustawiliśmy na 0 – ponieważ pierwszy element tablicy ma indeks 0.

Wykonujemy pętlę dopóki **`i < 3`**, to znaczy, że **`i`** nie może być równe 3.

Tablice i pętle

Jeżeli chcielibyśmy w prosty sposób wypisać wszystkie elementy tablicy w konsoli, możemy użyć pętli. Zobacz przykład poniżej:

```
int numbers[] = { 120, 3, 45 };
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + ", ");
}
```

Zauważ, że zmienną **`i`** ustawiliśmy na 0 – ponieważ pierwszy element tablicy ma indeks 0.

Wykonujemy pętlę dopóki **`i < 3`**, to znaczy, że **`i`** nie może być równe 3.

`numbers.length` – zwraca rozmiar tablicy;

Tablice i pętle

Jeżeli chcielibyśmy w prosty sposób wypisać wszystkie elementy tablicy w konsoli, możemy użyć pętli. Zobacz przykład poniżej:

```
int numbers[] = { 120, 3, 45 };
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + ", ");
}
```

Zauważ, że zmienną **i** ustawiliśmy na 0 – ponieważ pierwszy element tablicy ma indeks 0.

Wykonujemy pętlę dopóki **i < 3**, to znaczy, że **i** nie może być równe 3.

numbers.length – zwraca rozmiar tablicy;

wypisze: **120, 3, 45,**

Tablice i pętle

Po elementach tablicy możemy przejść używając pętli **`for`**, której schematyczny zapis jest następujący:

```
for (String variableName : tableName) {
    System.out.println(variableName);
}
```

Zmienna **`variableName`** musi być tego samego typu co elementy tablicy.

Zaletą jest niewątpliwie prosta konstrukcja, natomiast wadą – brak licznika.

Przykład:

```
String[] titles = new String[2];
for (String title : titles) {
    System.out.println(title);
}
```

Tablice i pętle

Po elementach tablicy możemy przejść używając pętli **`for`**, której schematyczny zapis jest następujący:

```
for (String variableName : tableName) {
    System.out.println(variableName);
}
```

Zmienna **`variableName`** musi być tego samego typu co elementy tablicy.

`String variableName` – typ i nazwa zmiennej, której będziemy używać w pętli, **`tableName`** – nazwa tablicy.

Zaletą jest niewątpliwie prosta konstrukcja, natomiast wadą – brak licznika.

Przykład:

```
String[] titles = new String[2];
for (String title : titles) {
    System.out.println(title);
}
```

Domyślne wartości

Przy tworzeniu tablicy bez jednoczesnej inicjalizacji wartości, każdy element otrzymuje domyślną wartość:

- **0** – gdy elementy są typu liczbowego,
- **false** – gdy elementy są typu logicznego,
- **null** – gdy elementy są typu obiektowego.

Przekroczenie zakresu

Zdarzyć się może, że odwołamy się do elementu, który w tablicy nie istnieje, np.:

```
String[] titles = new String[2];
System.out.println(titles[3]);
System.out.println(titles[2]);
```

Wynik jaki otrzymamy w konsoli:

```
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException:
  3 at testLoop.main(testLoop.java:33)
```

Przekroczenie zakresu

Zdarzyć się może, że odwołamy się do elementu, który w tablicy nie istnieje, np.:

```
String[] titles = new String[2];
System.out.println(titles[3]);
System.out.println(titles[2]);
```

Wynik jaki otrzymamy w konsoli:

```
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException:
  3 at testLoop.main(testLoop.java:33)
```

Zarówno jedna jak i druga linia spowoduje wystąpienie wyjątku. Pamiętajmy, że elementy tablicy numerowane są od 0.

Przekroczenie zakresu

Zdarzyć się może, że odwołamy się do elementu, który w tablicy nie istnieje, np.:

```
String[] titles = new String[2];
System.out.println(titles[3]);
System.out.println(titles[2]);
```

Wynik jaki otrzymamy w konsoli:

```
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException:
  3 at testLoop.main(testLoop.java:33)
```

Zarówno jedna jak i druga linia spowoduje wystąpienie wyjątku. Pamiętajmy, że elementy tablicy numerowane są od 0.

`testLoop` – nazwa klasy, w której wystąpił błąd, **`java:33`** – linia wystąpienia błędu.

Kopiowanie tablic

Tablice mają stały, niemodyfikowalny rozmiar, określony w sposób jawny podczas deklaracji:

```
String[] titles = new String[2];
```

lub podczas deklaracji z jednoczesną inicjalizacją:

```
String[] users = {"Alice", "John"};
```

Jednym ze sposobów na zwiększenie rozmiaru tablicy jest jej skopiowanie.

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

zwróci: 10

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

zwróci: 10

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Tablica, z której będziemy kopiować dane.

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

zwróci: 10

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Tablica, do której będziemy kopiować dane.

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

zwróci: 10

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Kopiowanie elementów jednej tablicy do drugiej.

Kopiowanie tablic

```
int[] tab = {1, 2, 3, 4};
int[] copyTab = tab;
tab[3] = 10;
System.out.println(copyTab[3]);
```

zwróci: 10

W ten sposób zamiast skopiować tablicę tworzymy dodatkową referencję.

Tablice również są obiektami. Zmienna **copyTab** jest referencją wskazującą na ten sam obiekt co **tab**.

Jednym ze sposobów na skopiowanie tablicy jest stworzenie tablicy o takim samym typie, skopiowanie elementów jednej do drugiej, a następnie ewentualne uzupełnienie dodatkowych wartości.

Przykład:

```
int[] tab1 = { 9, 21, 32, 42, 11 };
int[] copyTab1 = new int[7];
for (int i = 0; i < tab1.length; i++) {
    copyTab1[i] = tab1[i];
}
copyTab1[5] = 33;
copyTab1[6] = 44;
```

Uzupełnienie dodatkowych wartości.

Kopiowanie tablic

Kolejnym sposobem na skopiowanie tablicy jest wykorzystanie metody **`clone()`**:

```
int[] cloneArray = tab.clone();
```

Więcej na temat tej metody powiemy sobie omawiając obiekty.

Możemy również skorzystać z metod klasy **`java.util.Arrays`**:

```
int[] copyArray = Arrays.copyOf(tab, dataSize);
```

Kopiowanie tablic

Kolejnym sposobem na skopiowanie tablicy jest wykorzystanie metody **`clone()`**:

```
int[] cloneArray = tab.clone();
```

Więcej na temat tej metody powiemy sobie omawiając obiekty.

Możemy również skorzystać z metod klasy **`java.util.Arrays`**:

```
int[] copyArray = Arrays.copyOf(tab, dataSize);
```

`tab` – tablica, z której będziemy kopiować dane, **`dataSize`** – długość tablicy **`copyArray`**.

Arrays.copyOf

```
Arrays.copyOf(tab, dataSize);
```

Jeżeli drugi parametr będzie większy od rozmiaru kopiowanej tablicy dane zostaną automatycznie uzupełnione domyślną wartością dla danego typu.

Jeżeli będzie mniejszy – tylko część danych zostanie skopiowana.

W przypadku gdy kopiujemy dane do istniejącej już tablicy o określonym rozmiarze, rozmiar ten zostanie zmieniony (zmniejszony lub zwiększony).

```
int[] array1 = { 1, 2, 3 };
int[] array2 = new int[10];
array2 = Arrays.copyOf(array1, array1.length);
System.out.println(array2.length);
```

Arrays.copyOf

```
Arrays.copyOf(tab, dataSize);
```

Jeżeli drugi parametr będzie większy od rozmiaru kopiowanej tablicy dane zostaną automatycznie uzupełnione domyślną wartością dla danego typu.

Jeżeli będzie mniejszy – tylko część danych zostanie skopiowana.

W przypadku gdy kopiujemy dane do istniejącej już tablicy o określonym rozmiarze, rozmiar ten zostanie zmieniony (zmniejszony lub zwiększony).

```
int[] array1 = { 1, 2, 3 };
int[] array2 = new int[10];
array2 = Arrays.copyOf(array1, array1.length);
System.out.println(array2.length);
```

zwróci: 3

java.util.Arrays

Klasa **`java.util.Arrays`** udostępnia kilka użytecznych metod:

- **`Arrays.toString`** – zwraca elementy tablicy przekształcone na typ String,
- **`Arrays.copyOf(tab, dataSize)`** – kopiowanie,
- **`Arrays.fill(tab, element)`** – wypełnianie tablicy podanym elementem,
- **`Arrays.equals(tab1, tab2)`** – porównanie tablic,
- **`Arrays.sort(tab)`** – sortowanie tablicy.

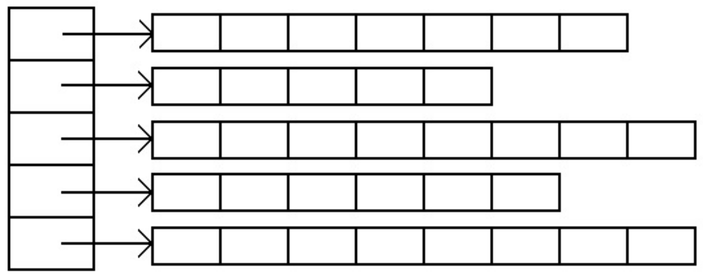
Zadania

Tablice wielowymiarowe

Tablice wielowymiarowe

- Dotychczas zajmowaliśmy się tylko tablicami jednowymiarowymi.
- Możemy definiować również tablice wielowymiarowe.
- W praktyce rzadko stosuje się tablice większe niż dwuwymiarowe.
- Tablice wielowymiarowe pozwalają na swego rodzaju zagnieżdżanie tablicy w tablicy.

Schematycznie moglibyśmy przedstawić tablicę wielowymiarową w poniższy sposób:



Tworzenie tablic wielowymiarowych

- Liczba wymiarów określana jest przez liczbę nawiasów **[]**.
- Tablice wielowymiarowe nazywa się również **tablicami tablic**.
- Rozmiary tablic podrzędnych nie muszą być równe.

```
int[][] tab = {
        { 1, 2, 3 },
        { 4, 5, 6, 7 } };
```

```
int[][] tab = new int[4][];
tab[0] = new int[3];
tab[1] = new int[2];
tab[2] = new int[1];
tab[3] = new int[9];
```

Tworzenie tablic wielowymiarowych

- Liczba wymiarów określana jest przez liczbę nawiasów **[]**.
- Tablice wielowymiarowe nazywa się również **tablicami tablic**.
- Rozmiary tablic podrzędnych nie muszą być równe.

```
int[][] tab = {
        { 1, 2, 3 },
        { 4, 5, 6, 7 } };
```

```
int[][] tab = new int[4][];
tab[0] = new int[3];
tab[1] = new int[2];
tab[2] = new int[1];
tab[3] = new int[9];
```

Występują 2 zestawy nawiasów – oznacza to tablicę dwuwymiarową.

Tworzenie tablic wielowymiarowych

- Liczba wymiarów określana jest przez liczbę nawiasów **[]**.
- Tablice wielowymiarowe nazywa się również **tablicami tablic**.
- Rozmiary tablic podrzędnych nie muszą być równe.

```
int[][] tab = {
        { 1, 2, 3 },
        { 4, 5, 6, 7 } };
```

```
int[][] tab = new int[4][];
tab[0] = new int[3];
tab[1] = new int[2];
tab[2] = new int[1];
tab[3] = new int[9];
```

Występują 2 zestawy nawiasów – oznacza to tablicę dwuwymiarową.

Tworzymy tablice podrzędne określając ich rozmiar.

Pętle

Tablice i pętle

Do wyświetlenia wszystkich elementów tablicy możemy skorzystać z podwójnej pętli **`for`**.

```
for (int i = 0; i < tab.length; i++) {
    for (int j = 0; j < tab[i].length; j++) {
        System.out.println(tab[i][j]);
    }
}
```

Pętle

Tablice i pętle

Do wyświetlenia wszystkich elementów tablicy możemy skorzystać z podwójnej pętli **`for`**.

```
for (int i = 0; i < tab.length; i++) {
    for (int j = 0; j < tab[i].length; j++) {
        System.out.println(tab[i][j]);
    }
}
```

`tab.length` – rozmiar tablicy zewnętrznej,

Pętle

Tablice i pętle

Do wyświetlenia wszystkich elementów tablicy możemy skorzystać z podwójnej pętli **`for`**.

```
for (int i = 0; i < tab.length; i++) {
    for (int j = 0; j < tab[i].length; j++) {
        System.out.println(tab[i][j]);
    }
}
```

`tab.length` – rozmiar tablicy zewnętrznej,

`tab[i].length` – rozmiar tablicy wewnętrznej.

Pętle

Tablice i pętle

Do wyświetlenia wszystkich elementów tablicy możemy skorzystać z podwójnej pętli **`for`**.

```
for (int i = 0; i < tab.length; i++) {
    for (int j = 0; j < tab[i].length; j++) {
        System.out.println(tab[i][j]);
    }
}
```

`tab.length` – rozmiar tablicy zewnętrznej,

`tab[i].length` – rozmiar tablicy wewnętrznej.

Wyświetlamy element pod indeksem **`i`** (w tablicy zewnętrznej)
oraz pod indeksem **`j`** (w tablicy wewnętrznej).

Korzystanie z tablic wielowymiarowych

- Jeżeli chcemy dostać się do jakiejś komórki z tablicy wielowymiarowej, musimy podać indeksy wszystkich wymiarów tej komórki.
- Indeks każdego wymiaru musi znajdować się w osobnych nawiasach kwadratowych!

```
int[][] tab = {
        { 1, 3, 4 },
        { 4, 5, 6 },
        { 7, 8, 9 } };
```

```
System.out.println(tab[0][0]);
System.out.println(tab[1][1]);
System.out.println(tab[2][2]);
```

Korzystanie z tablic wielowymiarowych

- Jeżeli chcemy dostać się do jakiejś komórki z tablicy wielowymiarowej, musimy podać indeksy wszystkich wymiarów tej komórki.
- Indeks każdego wymiaru musi znajdować się w osobnych nawiasach kwadratowych!

```
int[][] tab = {
        { 1, 3, 4 },
        { 4, 5, 6 },
        { 7, 8, 9 } };
```

```
System.out.println(tab[0][0]);
System.out.println(tab[1][1]);
System.out.println(tab[2][2]);
```

wyświetli: 1

Korzystanie z tablic wielowymiarowych

- Jeżeli chcemy dostać się do jakiejś komórki z tablicy wielowymiarowej, musimy podać indeksy wszystkich wymiarów tej komórki.
- Indeks każdego wymiaru musi znajdować się w osobnych nawiasach kwadratowych!

```
int[][] tab = {
        { 1, 3, 4 },
        { 4, 5, 6 },
        { 7, 8, 9 } };
```

```
System.out.println(tab[0][0]);
System.out.println(tab[1][1]);
System.out.println(tab[2][2]);
```

wyświetli: 1

wyświetli: 5

Korzystanie z tablic wielowymiarowych

- Jeżeli chcemy dostać się do jakiejś komórki z tablicy wielowymiarowej, musimy podać indeksy wszystkich wymiarów tej komórki.
- Indeks każdego wymiaru musi znajdować się w osobnych nawiasach kwadratowych!

```
int[][] tab = {
        { 1, 3, 4 },
        { 4, 5, 6 },
        { 7, 8, 9 } };
```

```
System.out.println(tab[0][0]);
System.out.println(tab[1][1]);
System.out.println(tab[2][2]);
```

wyświetli: 1

wyświetli: 5

wyświetli: 9

Zadania