Programowanie w języku

JAVA

- - - POLA - - -

**1. Pola statyczne:**

* 1. Pola statyczne różnią się od pól niestatycznych tym, że są one współdzielone przez wszystkie obiekty klasy, tzn. przynależą one do całej klasy, a nie konkretnie utworzonego obiektu.
  2. Wartości pól statycznych są współdzielone przez wszystkie obiekty klasy, w przeciwieństwie do pól niestatycznych, których własne egzemplarze ma każdy obiekt klasy, pola statyczne są tworzone jako pojedyncze wartości.
  3. Odnieść się do pola statycznego możemy powołując się na klasę, bez konieczności tworzenia obiektu. Przez utworzenie obiektu to też jest możliwe.

**2. Zastosowanie this dla pól:**

Jeżeli mamy w danej klasie zmienną znajdującą się wewnątrz metody oraz drugą zmienną znajdującą się poza metodą mającą taką samą nazwę, to żeby użyć zmiennej z poza metody wewnątrz metody, należy użyć słowa this.

**3. Zastosowanie this dla pól:**

Możemy zastosować this w przypadku gdy nazwiemy parametr konstruktora tak samo jak pole znajdujące się w tej samej klasie co ten konstruktor. W konstruktorze dokonujemy przypisania this.x = x i wtedy this.x odnosi się do pola poza konstruktorem. Czyli zastosowanie słowa kluczowego this znajdziemy w przypadku, gdy w jakiejś klasie posiadamy zmienne o takich samych nazwach jak parametry metod, czy konstruktorów. Nie zawsze chce nam się wymyślać coś nowego, albo po prostu ciężko znaleźć zamiennik.

- - - METODY - - -

**1. Nazewnictwo zmiennych:**

Różne metody mogą mieć zmienne, które tak samo się nazywają np. „int a” w kilku metodach.

**2. Działanie metody statycznej:**

2.1 Metody statyczne różnią się od metod niestatycznych tym, że nie mogą korzystać z pól i metod niestatycznych.

2.2 Możemy wywołać takie metody nie mając w ogóle jeszcze utworzonego żadnego obiektu danej klasy.

**3. Metoda w metodzie:**

W metodzie nie można stworzyć kolejnej metody.

**4. Metody o tej samej nazwie:**

W jednej klasie mogą być dwie metody o takiej samej nazwie przyjmujące parametry, jedynie wtedy gdy te parametry są od siebie różne (są różnego typu np. int oraz String), lub gdy jest ich różna ilość w tych metodach (w jednej metodzie int a w drugiej int oraz String).

- - - KLASY - - -

**1. Zastosowanie this w klasach:**

Znajdując się w określonej klasie i tworząc w niej pola, metody, konstruktory i zmienne lokalne czasami chcemy odnieść się do jakiegoś elementu znajdującego się wewnątrz tej klasy. Wtedy stosujemy przedrostek this.

**2. Klasa wewnętrzna:**

2.1 Jest to klasa, którą można utworzyć wewnątrz klasy zewnętrznej. Można w niej tworzyć metody i pola wewnętrzne oraz odnosić się i wywoływać pola i metody z klasy zewnętrznej. Nie można w klasie zewnętrznej odnosić się i wywoływać pól i metod z klasy wewnętrznej.

2.2 Nie można w klasie zewnętrznej odnosić się i wywoływać pól i metod z klasy wewnętrznej. Można odwoływać się i wywoływać pola i metody z klasy wewnętrznej w zewnętrznej jedynie przy użyciu obiektu. Można wywołać metodę zewnętrzną w wewnętrznej. Można przypisać wartość do pola zewnętrznego w klasie wewnętrznej.

2.3 Klasę wewnętrzną stosuje się tylko wtedy gdy wiemy, że będzie ona potrzebowała dostępu jedynie do klasy zewnętrznej, a nie do żadnej innej klasy. Wtedy nie musimy tworzyć kolejnej zbędnej klasy. Program wtedy nie rozrasta się niepotrzebnie.

**3. Klasa anonimowa:**

Jest to rodzaj klasy wewnętrznej i stosuje się ją gdy wiemy, że np. użyjemy obiektu tej klasy z jakimiś metodami jeden raz. Wtedy możemy uniknąć rozbudowywania programu poprzez tworzenie kolejnych klas.

**4. Klasa abstrakcyjne:**

4.1. Klasa abstrakcyjna to klasa zawierająca podstawowe funkcjonalności, podstawowe zachowania wspólne dla wszystkich klas dziedziczących po klasie abstrakcyjnej. Zawiera też metody abstrakcyjne, które są metodami przesłanianymi przez metody klas pochodnych. Dzieje się tak ponieważ dla różnych klas pochodnych te metody przesłaniające zawierają już różniące się od siebie funkcjonalności.

4.2 W main nie można stworzyć obiektu z klasy abstrakcyjnej, można jedynie go stworzyć z klasy dziedziczącej po klasie abstrakcyjnej.

4.3 W klasie abstrakcyjnej tworzymy oprócz metod zwykłych również metody abstrakcyjne, które nie mają „body”, służą tylko do tego by je przesłonić innymi metodami z klas pochodnych.

4.4 Metoda abstrakcyjna nie może mieć modyfikatora private ponieważ nie mogłaby być wtedy przesłonięta metodą z klasy pochodnej w procesie dziedziczenia. Aby można było dziedziczyć metodę musi mieć ona modyfikator dostępu w najgorszym razie protected.

4.4. Jest to taka klasa, którą tworzymy według wzorca:

public abstract class NazwaKlasyAbstrakcyjnej { }

- - - KONSTRUKTORY - - -

**1. Konstrukcja obiektu klasy bazowej i pochodnej:**

1.1 Konstrukcja obiektu klasy pochodnej zaczyna się od wywołania konstruktora klasy pochodnej, który wywołuje konstruktor swojej klasy bazowej.

1.2 Konstrukcja obiektu klasy bazowej polega na wywołaniu konstruktora tylko klasy bazowej.

**2. Dziedziczenie:**

Zarówno konstruktory w klasach bazowych jak i te w klasach pochodnych nie mogą być dziedziczone.

**3. Zastosowanie this w odniesieniu do konstruktora:**

Wewnątrz danej klasy zawierającej konstruktor pusty lub konstruktor z parametrami możemy odnieść się do tego konstruktora poprzez zastosowanie przedrostka this z nawiasami czyli np. dla konstruktora pustego użyjemy this().

**4. Zastosowanie super w odniesieniu do konstruktora:**

Wewnątrz danej klasy dziedziczącej możemy odnieść się do konstruktora klasy nadrzędnej poprzez zastosowanie przedrostka super z nawiasami czyli np. dla konstruktora z argumentem użyjemy super(argument).

- - - DZIEDZICZENIE - - -

**1. Przesłanianie metod:**

Jeżeli zastosujemy wyrażenie @Override „nad” metodą, która ma nazwę identyczną z nazwą metody występującej w klasie nadrzędnej, to umożliwi to wywoływanie metody dla obiektu z klasy, która dziedziczy po klasie nadrzędnej. Następuje przesłanianie jednej metody drugą metodą. Jeżeli pominiemy adnotację @Override, to powyższa procedura też będzie miała miejsce. Adnotacja ta pełni bardziej funkcję informacyjną dla programisty.

**2. Przesłanianie metod a polecenie super:**

Jeżeli stosujemy przesłanianie metody z klasy nadrzędnej metodą z klasy dziedziczącej, można wywołać metodę z klasy nadrzędnej wewnątrz metody z klasy dziedziczącej przy użyciu przedrostka super, czyli wywołanie metody np. super.nazwaMetody().

**3. Klasa nadrzędna:**

Określenie często stosowane dla klasy bazowej, z której dziedziczy klasa pochodna.

**4. Klasa pochodna:**

Określenie stosowane dla klasy dziedziczącej po klasie bazowej, nadrzędnej.

**5. Klasa object:**

Każda klasa domyślnie dziedziczy po klasie object.

**6. Rzutowanie obiektów:**

Jeżeli klasa dziedziczy po innej klasie to można ją zrzutować na dany typ. Czyli jeżeli mamy powołany do życia obiekt klasy pochodnej, to możemy tworząc obiekt klasy bazowej zrzutować go na obiekt klasy bazowej wg zasady: KlasaBazowa obiektKlasyBazowej = (KlasaBazowa) obiektKlasyPochodnej.

- - - WĄTKI - - -

**1. Klasa Thread:**

* 1. Wątek tworzymy poprzez utworzenie klasy, która dziedziczy po klasie Thread, według schematu: public class nazwaKlasy extends Thread { }
  2. Kolejny krok to utworzenie w klasie dziedziczącej po thread metody przesłaniającej run, w której zawarty będzie realizowany kod, według schematu:

@Override

public void run( )

{ kod do zrealizowania w wątku }

* 1. Uruchomienie wątku w main odbywa się przy użyciu polecenia start, po utworzeniu wpierw obiektu z klasy dziedziczącej po Thread, według schematu:

KlasaDziedziczacaPoThread nazwaObiektu = new KlasaDziedziczacaPoThread( );

Obiekt.start( );

**2. Interfejs Runnable:**

* 1. Różni się od klasy Thread tym, że dana klasa może implementować interfejs Runnable oraz inne interfejsy jednocześnie. Natomiast jeżeli chcielibyśmy żeby dana klasa dziedziczyła klasę Thread, to może odziedziczyć ją tylko jedną, bo nie można dziedziczyć kilku klas.
  2. Stosowanie wątków poprzez użycie interfejsu Runnable wykonujemy począwszy od wpisania następującego kodu w danej klasie:

public class NazwaKlasy implements Runnable

{

@Override

Public void run( ){ Kod wykonywany w wątku }

}

* 1. Kolejny krok to wpisanie w klasie main poniższego kodu:

Thread nazwaWatku = new Thread(new NazwaKlasy);

nazwaWatku.start( );

**3. Interfejs Runnable z klasą anonimową:**

3.1 Stosuje się takie rozwiązanie jeżeli nie ma konieczności tworzenia kolejnej dużej klasy dla danego wątku, a jedynie chcemy w klasie main wykonać raz jakąś instrukcje, ale na oddzielnym wątku.

3.2 Przykładowy kod w klasie main:

Thread nazwaWatku = new Thread(new Runnable( ) {

@Override

public void run( ) { Realizowany kod }

});

nazwaWatku.start( );

**4. Interfejs Runnable z wątkiem bez przypisania zmiennej:**

4.1 Stosuje się takie rozwiązanie jeżeli nie ma konieczności tworzenia kolejnej dużej klasy dla danego wątku, a jedynie chcemy w klasie main wykonać raz jakąś instrukcje, ale na oddzielnym wątku. Wątek stworzony jest tak, że nie przypisujemy żadnej zmiennej.

4.2 Przykładowy kod w klasie main:

new Thread(new Runnable( ) {

@Override

public void run( ) { Realizowany kod }

}).start( );

**5. Uśpienie wątku sleep:**

5.1 Sleep umożliwia uśpienie wykonywania kodu danego wątku na określony czas. Występuje tu wyjątek typu check, który należy obsłużyć przez try/catch.

5.2 Kod wygląda następująco i może być umieszczony w różnych miejscach, co będzie powodować różne opóźnienia w zależności gdzie go umieścimy:

Try

{

Thread.sleep(czas w milisekundach);

}

catch (InterruptedException e) { }

**6. Wstrzymanie wykonywania kodu join:**

6.1 Join umożliwia zatrzymanie wykonywania kodu, który znajduje się po join aż do chwili gdy znajdujący się przed join wątek wykona się do końca.

6.2 Join razem z thread tworzy wyjątek typu check, który musi być obsłużony przez try/catch. Przykładowy kod: try {nazwawątku.join( );} catch (InterruptedException e) { }

**7. Sprawdzenie czy wątek żyje isAlive:**

7.1 Aby sprawdzić czy wątek, który rozpoczął wykonywanie się jeszcze żyje lub czy już skończył pracę, należy użyć polecenia nazwaWątku.isAlive( )

7.2 Można umieścić isAlive przed i po join wtedy widzimy dokładnie kiedy wątek rozpoczął i skończył pracę.

- - - WIELKOŚĆ ZNAKÓW - - -

**1. static final String:**

Jeżeli mamy „static final String NAZWA\_POLA” to nazwę pola zapisujemy w konwencji jak pokazano w powyższym przykładzie.

**2. camelCase:**

System notacji ciągów tekstowych, w którym kolejne wyrazy pisane są łącznie, rozpoczynając każdy następny wielką literą (oprócz pierwszego). Na przykład: nazwaZmiennej, kolejnaNazwaZmiennej.

Znany jest również pod nazwą bumpyCaps.

**3. PascalCase:**

System notacji ciągów tekstowych, w którym kolejne wyrazy pisane są łącznie, rozpoczynając każdy z nich wielką literą. Na przykład: BackColor, FatalError. Zapis PascalCase, w przeciwieństwie do notacji camelCase, zawsze rozpoczyna się wielką literą. Znany jest również pod nazwą UpperCamelCase.

- - - POJĘCIA I DEFINICJE - - -

**1. Redundancja:**

Nadmiarowość kodu w stosunku do ilości jaka jest niezbędna do uzyskania pożądanego efektu.

- - - KOLEJNOŚĆ W PISNIU KODU - - -

I. Pola publiczne piszemy najpierw, potem prywatne.

II. Pola final piszemy najpierw, potem pola bez final.

III. Najpierw piszemy wszystkie pola, potem piszemy konstruktor, potem piszemy metody.

IV. Jeżeli w metodzie public odwołujemy się do metody private, to ta metoda private musi w kodzie znajdować się bezpośrednio pod metodą public.