Politechnika Wrocławska

Platformy Programistyczne .NET i Java

Programowanie w języku JAVA

Dr inż. Radosław Idzikowski

HR EXCELLENCE IN RESEARCH

Politechnika Wrocławska

Agenda

1. Obsługa wyjątków
2. Klasy abstrakcyjne
3. Interfersjy
4. Klasy wyliczeniowe
5. Wyrażenia lambda
6. Programowanie generyczne
7. Kolekcje

Obsługa wyjątków

Można wyróżnić dwa rodzaje wyjątków:

- **niekontrolowane** -– kompilator nie sprawdza, czy zadbano o obsługę tych błędów, np. dostęp do referencji `null`,
- **kontrolowane** -– kompilator sprawdza, czy napisano procedurę do ich obsługi.

W bloku `try` należy umieścić instrukcje, które mogą spowodować wyjątek. W bloku `catch` należy umieścić kod obsługujący wyjątek.

Obsługa wyjątków

```
try
{
    // instructions
}
catch(Exception e)
{
    // handle exception
}
finally
{
    // final action
}
```

Obsługa wyjątków

```
class TestException03
{
  public static void main(String[] args)
  {
    String s=null;
    System.out.println(s.length());
  }
}
```

```
$ javac TestException03.java
$ java TestException03
Exception in thread "main" java.lang.NullPointerException
       at TestException03.main(TestException03.java:10)
```

Obsługa wyjątków

```
class TestException04
{
  public static void main(String[] args)
  {
    Scanner inf = new Scanner(Paths.get(args[0]));
    while(inf.hasNextLine())
    {
      System.out.println(inf.nextLine());
    }
        inf.close();
  }
}
```

```
TestException04.java:9: error: unreported exception IOException; must be caught or declared to
    Scanner inf = new Scanner(Paths.get(args[0]));
                              ^
1 error
```

Obsługa wyjątków

- `throws` jest słowem kluczowym, które jest używane w sygnaturze metody w celu wskazania, że ta metoda może zgłaszać jeden z wymienionych wyjątków.
- Program wywołujący te metody musi obsłużyć wyjątek za pomocą bloku `try-catch` lub oddelegować obsługę wyjątku do wywołującego.

Obsługa wyjątków

Jeśli kod w bloku `try` zgłosi wyjątek typu z klauzuli `catch`, to:

1. pozostały kod w bloku `try` zostanie pominięty,
2. zostanie wykonany kod z bloku `catch` obsługujący zgłoszony wyjątek,
3. zostanie wykonany kod w bloku `finally`.

Obsługa wyjątków

Klauzuli `finally` można użyć bez klauzuli `catch`.

```
InputStream in = new FileInputStream(args[0]);
try
{
    // instructions
}
finally
{
    in.close();
}
```

Instrukcja `in.close()` zostanie wykonana bez względu na to czy wystąpi wyjątek.

Obsługa wyjątków

Wcelu ułatwienia pracy z zasobami można wykorzystać klauzulę `try` z zasobami.

```
try(InputStream in = new FileInputStream(args[0]))
{
    // instructions
}
```

Jeśli działanie bloku kodu zakończy się normalnie lub wystąpi wyjątek, to wywołana zostanie metoda `in.close()` – jak gdyby zdefiniowana była klauzula `finally`

Obsługa wyjątków

Jeżeli w programie może wystąpić problem, do którego nie pasuje żadna ze standardowych klas wyjątków, to należy utworzyć klasę, która dziedziczy po klasie `Exception` lub jednej z jej podklas.

```
class FileExtException extends IOException
{
    public FileExtException() {}
    public FileExtException(String msg)
    {
        super(msg);
    }
}
```

Obsługa wyjątków

Wskazówki dla stosowania wyjątków:

- oddzielanie od siebie bloków `try-catch` i `try-finally`,
- wyjątki nie powinny zastępować prostych testów,
- nie należy rozdrabniać się,
- nie należy wyciszać/ukrywać wyjątków,
- właściwe wykorzystanie hierarchii wyjątków,
- „wczesna generacja, późne przechwycenie”.

Klasy abstrakcyjne

- Klasa musi być abstrakcyjna jeśli zawiera co najmniej jedną metodę abstrakcyjną.
- Klasa abstrakcyjna może mieć zarówno metody abstrakcyjne (bez implementacji), jak i metody zdefiniowane (z implementacją).
- Klasa abstrakcyjna może dziedziczyć tylko jedną klasę.
- Deklaruje się ją przy użyciu słowa kluczowego `abstract`.
- Może mieć zmienne instancyjne (pola) i zmienne statyczne.
- Może zawierać konstruktor.
- Metody w klasie abstrakcyjnej mogą mieć różne modyfikatory dostępu (`public`, `protected`, `private`).
- Używane, gdy istnieje potrzeba zapewnienia wspólnego zachowania (implementacji) dla grupy klas, oprócz wymuszenia implementacji pewnych metod.

Klasy abstrakcyjne

```
abstract class Shape {
    abstract void draw();
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing␣a␣Circle");
    }
}
class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing␣a␣Rectangle");
    }
}
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();
        circle.draw();
        rectangle.draw();
    }
}
```

Klasy abstrakcyjne

1. Klasa Abstrakcyjna `Shape`: Definiuje metodę `draw()`, która musi być zaimplementowana przez klasy dziedziczące.
2. Klasa `Circle`: Implementuje metodę `draw()`, która wypisuje `"Drawing a Circle"`.
3. Klasa `Rectangle`: Implementuje metodę `draw()`, która wypisuje `"Drawing a Rectangle"`.
4. Klasa `Main`: Zawiera metodę `main`, która tworzy obiekty `Circle` i `Rectangle`, a następnie wywołuje ich metody `draw()`.

Uruchomienie tego kodu spowoduje wyświetlenie:

```
Drawing a Circle
Drawing a Rectangle
```

Ten przykład pokazuje, jak można wykorzystać klasy abstrakcyjne w `Javie` do definiowania metod, które muszą być zaimplementowane przez klasy dziedziczące.

Interfersjy

- Interfejs może zawierać tylko metody abstrakcyjne (do `Javy 7`), metody domyślne, statyczne oraz prywatne (od `Javy 8`).
- Klasa może implementować wiele interfejsów.
- Deklaruje się go przy użyciu słowa kluczowego `interface`.
- Może mieć tylko stałe (pola `final static`) – do `Javy 7`, od `Javy 8` może mieć także metody domyślne oraz statyczne.
- Nie może zawierać konstruktora.
- Wszystkie metody są domyślnie `public` i abstrakcyjne (do `Javy 7`).
- Od `Javy 8`, interfejsy mogą mieć metody zdefiniowane (metody domyślne oraz statyczne).
- Używane, gdy różne klasy muszą implementować te same metody, ale nie mają wspólnego przodka.
- Daje możliwość osiągnięcia wielodziedziczenia typu przez implementację wielu interfejsów.

Interfersjy

```
interface Drawable {
    void draw();
}
class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing␣a␣Circle");
    }
}
class Rectangle implements Drawable {
    public void draw() {
        System.out.println("Drawing␣a␣Rectangle");
    }
}
public class Main {
    public static void main(String[] args) {
        Drawable circle = new Circle();
        Drawable rectangle = new Rectangle();
        circle.draw();
        rectangle.draw();
    }
}
```

Interfersjy

1. Interfejs `Drawable`: Definiuje metodę `draw()`, którą muszą zaimplementować klasy implementujące ten interfejs.
2. Klasa `Circle`: Implementuje interfejs `Drawable` i jego metodę `draw()`, która wypisuje `"Drawing a Circle"`.
3. Klasa `Rectangle`: Implementuje interfejs `Drawable` i jego metodę `draw()`, która wypisuje `"Drawing a Rectangle"`.
4. Klasa `Main`: Zawiera metodę `main`, która tworzy obiekty `Circle` i `Rectangle`, a następnie wywołuje ich metody `draw()`.

Uruchomienie tego kodu spowoduje wyświetlenie:

```
Drawing a Circle
Drawing a Rectangle
```

Ten przykład pokazuje, jak można wykorzystać interfejsy w `Javie` do definiowania metod, które muszą być zaimplementowane przez klasy implementujące ten interfejs.

Klasa abstrakcyjna a interfejs

- **Klasa abstrakcyjna**: Używamy, gdy chcemy stworzyć bazową klasę z wspólnymi zachowaniami i wymusić implementację pewnych metod w klasach potomnych. Obsługuje jedynie pojedyncze dziedziczenie.
- **Interfejs**: Używamy, gdy chcemy zdefiniować kontrakt, który różne klasy muszą zaimplementować, niezależnie od ich hierarchii dziedziczenia. Obsługuje wielodziedziczenie przez implementację wielu interfejsów.

Klasy wyliczeniowe

```
class enum Color
{
    RED, GREEN, BLUE
}
```

Zdefiniowany powyżej typ to klasa. Posiada ona dokładnie trzy egzemplarze -– nie można tworzyć jej nowych obiektów.

Klasy wyliczeniowe

Wszystkie typy są podklasami klasy `Enum` po której dziedziczą metody:

- `toString` – zwraca nazwę stałej wyliczeniowej,
- `valueOf` – zwraca stałą wyliczeniową dla podanego ciągu znaków,
- `values` – zwraca wszystkie wartości wyliczenia,
- `ordinal` – zwraca położenie stałej wyliczeniowej w deklaracji `enum`.

Wyrażenia lambda

Wyrażenia lambda w Javie zostały wprowadzone w wersji Java 8 i pozwalają na bardziej zwięzłe i czytelne wyrażenie instancji interfejsów funkcyjnych (interfejsów z jedną metodą abstrakcyjną). Dzięki wyrażeniom lambda można pisać krótszy i bardziej czytelny kod, szczególnie przy pracy z kolekcjami czy strumieniami (streams).

Wyrażenie lambda składa się z:

- listy parametrów,
- strzałki (->),
- ciała wyrażenia.

```
(parameters) -> expression
(parameters) -> { statements; }
```

Lambda

```
class TestLambda02
{
  public static void main(String[] args)
  {
    var ints = new ArrayList<Integer>();
    ints.add(1);
    ints.add(2);
    ints.add(3);
    ints.add(4);
    ints.forEach(n -> System.out.println(n));
  }
}
```

Lambda

```
class TestLambda
{
  public static void main(String[] args)
  {
    Runnable r = () ->
      System.out.println("Hello␣from␣lambda");
    r.run();
  }
}
```

Programowanie generyczne

Programowanie generyczne (ang. `generic programming`) to pisanie kodu źródłowego, który pozwala na tworzenie obiektów wielu różnych typów. Klasa `ArrayList` stanowi przykład programowania generycznego.

```
public class Pair<U, S>
{
    private U first;
    private S second;
    ...
}
```

Programowanie generyczne

Zmienne typowe mają krótkie nazwy zaczynające się wielką literą:

- `T` – typ elementu kolekcji,
- `K` i `V` – typy kluczy i wartości tablic,
- `T`, `U`, `S` – dowolny typ.

Programowanie generyczne

Metody generyczne mogą być definiowane zarówno w zwykłych klasach, jak i klasach generycznych.

```
private static <T> boolean isEqual(T a, T b)
{
    return a.equals(b);
}
// ...
System.out.println(isEqual("1", "3"));
Pair<Integer, String> p1 = new Pair<>(1, "one");
Pair<Integer, String> p2 = new Pair<>(2, "two");
System.out.println(isEqual(p1, p2));
```

Programowanie generyczne

Zalety programowania generycznego:

- powtórne wykorzystanie kodu źródłowego – raz napisana klasa/metoda/interfejs może zostać wykorzystana dla dowolnego typu,
- bezpieczeństwo typów – silniejsza kontrola typów podczas kompilacji,
- brak konieczności indywidualnego rzutowania typów,
- implementacja generycznych algorytmów operujących na kolekcjach różnych typów.

Programowanie generyczne

Ograniczenia programowania generycznego:

- nie można tworzyć instancji obiektów generycznych dla typów prostych,
- nie można tworzyć instancji zmiennych typowych,
- nie można używać zmiennych typowych w statycznych kontekstach klas generycznych,
- nie można rzutować typów generycznych oraz wykorzystywać `instanceof`,
- nie można tworzyć tablic typów generycznych.

Typ wieloznaczny

Typ wieloznaczny stanowi rozwiązanie problemu sztywnych systemów typów generycznych:

- w typie wieloznacznym (`wildcard type`) parametr typu może się zmieniać,
- `ArrayList<? extends Task>` – parametr typu jest podklasą klasy `Task`,
- `ArrayList<? super SubTask>` – parametr typu jest ograniczony do wszystkich nadtypów typu `SubTask`,
- `ArrayList<?>` – brak ograniczeń.

Typ wieloznaczny

```java
public class WildcardExample {
    public static void printList(List<?> list) {
        for (Object elem : list) {
            System.out.println(elem);
        }
    }

    public static void main(String[] args) {
        List<String> stringList = List.of("one", "two", "three");
        List<Integer> integerList = List.of(1, 2, 3);

        printList(stringList);  // Output: one, two, three
        printList(integerList); // Output: 1, 2, 3
    }
}
```

Typy wieloznaczne w Javie pozwalają na bardziej elastyczne i uniwersalne kodowanie, szczególnie w kontekście programowania ogólnego. Umożliwiają one operowanie na kolekcjach i innych strukturach danych bez sztywnego określania typów, co zwiększa ogólność i reużywalność kodu.

Kolekcje

System kolekcji Javy dzieli się na warstwę interfejsów oraz warstwę implementacji. Kluczowe znaczenie ma interfejs `Collection`, który zawiera m.in. metody:

- `add` – dodaje element do kolekcji,
- `iterator` – zwraca obiekt implementujący interfejs `Iterator`, który pozwala na dostęp do kolejnych elementów kolekcji,
- `remove` – usuwa element z kolekcji,
- `removeIf` – usuwa elementy z kolekcji spełniające zadane warunki,
- `size` – zwraca liczbę elementów w kolekcji,
- `clear` – usuwa wszystkie elementy z kolekcji,
- `contains` – zwraca `true` jeśli kolekcja zawiera dany element.

Inne interfejsy systemu kolekcji to `Map`, `List`, `Set`, `SortedSet`, `SortedMap`, `NavigableSet`, `NavigableMap`.

Kolekcje — iteratory

Obiekty implementujące interfejs `Iterator` pozwalają na dostęp do kolejnych elementów kolekcji. Główne metody interfejsu `Iterator` to:

- `next` – pozwala na przejście przez kolejne elementy kolekcji,
- `hasNext` – zwraca `true` jeśli w kolekcji są jeszcze jakieś elementy,
- `remove` – usuwa element zwrócony przez ostatnie wywołanie metody `next`.

Metody `remove` nie można wywołać jeśli wcześniej nie wywołano metody `next`.

Kolekcje — iteratory

```
Iterator<String> iter = words.iterator();
while(iter.hasNext())
{
    String w = iter.next();
    System.out.println(w);
}
iter = words.iterator();
iter.forEachRemaining(element->
    {
        System.out.println(element);
    });
```

Kolekcje

- `ArrayList` – indeksowana lista o dynamicznym rozmiarze,
- `LinkedList` – uporządkowana lista z możliwością szybkiego wstawiania i usuwania elementów,
- `ArrayDeque` – lista cykliczna bez początku i końca,
- `HashSet` – nieuporządkowana lista unikatowych obiektów,
- `TreeSet` – uporządkowany zbiór,
- `EnumSet` – zbiór wartości typu wyliczeniowego,
- `LinkedHashSet` – zbiór zachowujący kolejność wstawianych elementów.

Kolekcje

- `PriorityQueue` – kolekcja pozwalająca na szybkie usunięcie najmniejszego elementu,
- `HashMap` – struktura danych przechowująca pary klucz–wartość,
- `TreeMap` – słownik sortujący klucze,
- `EnumMap` – słownik, w którym klucze to typy wyliczeniowe,
- `LinkedHashMap` – słownik pamiętający kolejność wstawianych elementów,
- `WeakHashMap` – słownik, którego elementy mogą zostać usunięte przez `garbage collector`,
- `IdentityHashMap` – słownik, który przechowuje klucze porównywane za pomocą operatora $==$, a nie metody `equals`.

Kolekcje — algorytmy

- sortowanie:

```
Collections.sort(words);
Collections.sort(words, Comparator.reverseOrder());
Collections.sort(words, (s1, s2)->
    {
        return s1.length()- s2.length();
    });
```

- tasowanie:

```
Collections.shuffle(words);
```

Kolekcje — algorytmy

- wyszukiwanie binarne:

```
Collections.sort(words);
int i = Collections.binarySearch(words, "claqueur");
```

- proste algorytmy

```
words.removeIf(w-> w.length() > 10);
words.replaceAll(String::toUpperCase);
```