

Podstawy Java – dzień 2

v3.1

Napisy

Typ String

Klasa String reprezentuje ciąg znaków:

```
String text = "Ala ma kota";  
String name = "John";  
String age = "20";
```

Zwróć uwagę, że napisy umieszczamy w cudzysłowie ("Ala ma kota"), natomiast apostrofy służą do nadawania wartości dla typu char:

```
char myChar = 'a';
```

Stringi możemy dodawać do siebie przy pomocy znaku:

+

Operacja łączenia łańcuchów znaków jest nazywana **konkatenacją**.

```
name + " Travalta";  
// otrzymamy wynik:  
"John Travalta"
```

String

Inicjalizacja zmiennej typu String może przebiegać na dwa sposoby:

```
String variableName = new String("name");
```

lub:

```
String variableName = "name";
```

Są to konstrukcje równoważne.

String

Modyfikacje obiektów typu String

- Obiekty klasy **String** są niemodyfikowalne.
- Gdy nadamy zmiennej wartość – nie może ona zostać zmieniona w sposób bezpośredni.
- Wartości mogą być zmienione tylko w wyniku przypisania.

Rozważmy następujący przykład:

```
String text = "abc";  
System.out.println(text);  
text.toUpperCase();  
System.out.println(text);
```

String

Modyfikacje obiektów typu String

- Obiekty klasy **String** są niemodyfikowalne.
- Gdy nadamy zmiennej wartość – nie może ona zostać zmieniona w sposób bezpośredni.
- Wartości mogą być zmienione tylko w wyniku przypisania.

Rozważmy następujący przykład:

```
String text = "abc";  
System.out.println(text);  
text.toUpperCase();  
System.out.println(text);
```

zwróci: **abc**

String

Modyfikacje obiektów typu String

- Obiekty klasy **String** są niemodyfikowalne.
- Gdy nadamy zmiennej wartość – nie może ona zostać zmieniona w sposób bezpośredni.
- Wartości mogą być zmienione tylko w wyniku przypisania.

Rozważmy następujący przykład:

```
String text = "abc";  
System.out.println(text);  
text.toUpperCase();  
System.out.println(text);
```

zwróci: **abc**

również zwróci: **abc** (!)

String

Modyfikacje obiektów typu String

- Obiekty klasy **String** są niemodyfikowalne.
- Gdy nadamy zmiennej wartość – nie może ona zostać zmieniona w sposób bezpośredni.
- Wartości mogą być zmienione tylko w wyniku przypisania.

Rozważmy następujący przykład:

```
String text = "abc";  
System.out.println(text);  
text.toUpperCase();  
System.out.println(text);
```

zwróci: **abc**

również zwróci: **abc** (!)

Dzieje się tak dlatego że metoda **toUpperCase()** nie zmienia bieżącego obiektu, tylko tworzy nowy obiekt. Jeśli chcemy go później wykorzystać w kodzie – należy przypisać ten obiekt do zmiennej.

String

Metody klasy String

Klasa **String** zawiera szereg przydatnych metod np.:

```
String text = "AbCdEfAbCdEf";  
String subText = text.substring(1);  
char charAt = text.charAt(0);  
String lowerText = text.toLowerCase();  
String upperText = text.toUpperCase();
```

String

Metody klasy String

Klasa **String** zawiera szereg przydatnych metod np.:

```
String text = "AbCdEfAbCdEf";  
String subText = text.substring(1);  
char charAt = text.charAt(0);  
String lowerText = text.toLowerCase();  
String upperText = text.toUpperCase();
```

Metoda zwraca część łańcucha znakowego, zaczynający się na pozycji o wskazanym indeksie:

zwróci: **bCdEfAbCdEf**.

String

Metody klasy String

Klasa **String** zawiera szereg przydatnych metod np.:

```
String text = "AbCdEfAbCdEf";  
String subText = text.substring(1);  
char charAt = text.charAt(0);  
String lowerText = text.toLowerCase();  
String upperText = text.toUpperCase();
```

Metoda zwraca część łańcucha znakowego, zaczynający się na pozycji o wskazanym indeksie:

zwróci: **bCdEfAbCdEf**.

Metoda zwraca znak z pozycji o określonym indeksie:

zwróci: **A**.

String

Metody klasy String

Klasa **String** zawiera szereg przydatnych metod np.:

```
String text = "AbCdEfAbCdEf";  
String subText = text.substring(1);  
char charAt = text.charAt(0);  
String lowerText = text.toLowerCase();  
String upperText = text.toUpperCase();
```

Metoda zwraca część łańcucha znakowego, zaczynający się na pozycji o wskazanym indeksie:

zwróci: **bCdEfAbCdEf**.

Metoda zwraca znak z pozycji o określonym indeksie:

zwróci: **A**.

Metoda zamieniająca litery na małe:

zwróci: **abcdefabcdef**.

String

Metody klasy String

Klasa **String** zawiera szereg przydatnych metod np.:

```
String text = "AbCdEfAbCdEf";  
String subText = text.substring(1);  
char charAt = text.charAt(0);  
String lowerText = text.toLowerCase();  
String upperText = text.toUpperCase();
```

Metoda zwraca część łańcucha znakowego, zaczynający się na pozycji o wskazanym indeksie:

zwróci: **bCdEfAbCdEf**.

Metoda zwraca znak z pozycji o określonym indeksie:

zwróci: **A**.

Metoda zamieniająca litery na małe:

zwróci: **abcdefabcdef**.

Metoda zamieniająca litery na wielkie:

zwróci: **ABCDEFABCDEF**.

String

Metody klasy String

```
String text = "AbCdEfAbCdEf";  
int textLength = text.length();  
String replaceFirstText = text.replaceFirst("b", "BB");  
String replaceAllText = text.replaceAll("b", "BBB");  
String concatText = text.concat("XyZ");
```

String

Metody klasy String

```
String text = "AbCdEfAbCdEf";  
int textLength = text.length();  
String replaceFirstText = text.replaceFirst("b", "BB");  
String replaceAllText = text.replaceAll("b", "BBB");  
String concatText = text.concat("XyZ");
```

length() – zwraca długość napisu.

String

Metody klasy String

```
String text = "AbCdEfAbCdEf";  
int textLength = text.length();  
String replaceFirstText = text.replaceFirst("b", "BB");  
String replaceAllText = text.replaceAll("b", "BBB");  
String concatText = text.concat("XyZ");
```

length() – zwraca długość napisu.

Zamiana pierwszego wystąpienia dopasowanego tekstu (zamieni pierwsze wystąpienie **b** na **BB**).

String

Metody klasy String

```
String text = "AbCdEfAbCdEf";  
int textLength = text.length();  
String replaceFirstText = text.replaceFirst("b", "BB");  
String replaceAllText = text.replaceAll("b", "BBB");  
String concatText = text.concat("XyZ");
```

length() – zwraca długość napisu.

Zamiana pierwszego wystąpienia dopasowanego tekstu (zamieni pierwsze wystąpienie **b** na **BB**).

Zamiana wszystkich wystąpień dopasowanego tekstu (zamieni wszystkie wystąpienia **b** na **BBB**).

String

Metody klasy String

```
String text = "AbCdEfAbCdEf";  
int textLength = text.length();  
String replaceFirstText = text.replaceFirst("b", "BB");  
String replaceAllText = text.replaceAll("b", "BBB");  
String concatText = text.concat("XyZ");
```

length() – zwraca długość napisu.

Zamiana pierwszego wystąpienia dopasowanego tekstu (zamieni pierwsze wystąpienie **b** na **BB**).

Zamiana wszystkich wystąpień dopasowanego tekstu (zamieni wszystkie wystąpienia **b** na **BBB**).

Wykonanie operacji łączenia znaków. Wynik tej operacji będzie tożsamy z wykonaniem połączenia przy użyciu znaku plus ("+").

String

Metody klasy String

Powszechnie stosowaną i przydatną metodą jest metoda `trim()`:

```
String trimText = " text with spaces ";  
String newText = trimText.trim();  
System.out.println(newText);
```

Usuwa białe znaki z początku i końca napisu.

Ciekawą nowością **Javy** od wersji **1.8** jest metoda `join()`:

```
String joinText =  
String.join("-", "Java", "8", "News");
```

String

Metody klasy String

Powszechnie stosowaną i przydatną metodą jest metoda **trim()**:

```
String trimText = " text with spaces ";  
String newText = trimText.trim();  
System.out.println(newText);
```

Usuwa białe znaki z początku i końca napisu.

Rezultat: **"text with spaces"**

Ciekawą nowością **Javy** od wersji **1.8** jest metoda **join()**:

```
String joinText =  
String.join("-", "Java", "8", "News");
```

String

Metody klasy String

Powszechnie stosowaną i przydatną metodą jest metoda **trim()**:

```
String trimText = " text with spaces ";  
String newText = trimText.trim();  
System.out.println(newText);
```

Usuwa białe znaki z początku i końca napisu.

Rezultat: **"text with spaces"**

Ciekawą nowością **Javy** od wersji **1.8** jest metoda **join()**:

```
String joinText =  
String.join("-", "Java", "8", "News");
```

"- -" – napis łączący.

Java, 8, News – napisy do połączenia.

Wynik: Java--8--News

String

Metody klasy String

Pełną listę możliwych operacji klasy **String** znajdziemy w dokumentacji:

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

Zadania

Wykonaj zadania z działu

Napisy

Tworzenie i podział napisów

Tworzenie napisów

Klasa `StringBuilder`

Klasa ta pozwala na modyfikowanie przechowywanych łańcuchów znaków.

Korzystamy z niej również do budowania tekstów.

Jest to wydajny sposób na modyfikację napisów.

Obiekty tej klasy to bufory, które dynamicznie możemy wypełniać napisami.

```
StringBuilder sb = new StringBuilder();
sb.append("text 1 ");
sb.append("text 2 ");
sb.append(10);
sb.append(' ');
sb.append("text 3");
System.out.println(sb.toString());
```

Tworzenie napisów

Klasa `StringBuilder`

Klasa ta pozwala na modyfikowanie przechowywanych łańcuchów znaków.

Korzystamy z niej również do budowania tekstów.

Jest to wydajny sposób na modyfikację napisów.

Obiekty tej klasy to bufory, które dynamicznie możemy wypełniać napisami.

```
StringBuilder sb = new StringBuilder();  
sb.append("text 1 ");  
sb.append("text 2 ");  
sb.append(10);  
sb.append(' ');  
sb.append("text 3");  
System.out.println(sb.toString());
```

`toString()` – metoda, która zwraca nam zawartość obiektu **`sb`** jako napis:

wyświetli: **text 1 text 2 10 text 3**

Tworzenie napisów

Klasa `StringBuilder`

Metoda **`append()`** zwraca obiekt, możliwe jest więc wykonywanie jej w następujący sposób:

```
sb.append("text 1 ").append("text 2 ").append(10).append(' ').append("text 3");
```

Wykonywanie operacji w łańcuchu (ang. **chaining**) jest możliwe tylko w przypadku użycia metod, które zwracają przetworzony element.

Możemy korzystać z operacji chainingu, gdy musimy połączyć dużą liczbę napisów, np. w pętli:

```
StringBuilder sbLoop = new StringBuilder();  
for (int i = 0; i < 1000; i++) {  
    sbLoop.append("iteration " + i).append('\n');  
}  
System.out.println(sbLoop.toString());
```

Tworzenie napisów

String vs StringBuilder

- Obiekty klasy **String** są niemodyfikowalne.
- Operacje na obiektach typu String powodują utworzenie nowego obiektu.
- Klasa **String** służy do reprezentowania napisów.
- Klasa **StringBuilder** służy do budowania napisów.

Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";  
StringBuilder sb = new StringBuilder();  
sb.append(text);  
sb.reverse();  
System.out.println(sb.toString());
```

String

```
String text = "AbCdEfAbCdEf";  
System.out.println(text);  
text.toLowerCase();  
System.out.println(text);
```

Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";  
StringBuilder sb = new StringBuilder();  
sb.append(text);  
sb.reverse();  
System.out.println(sb.toString());
```

Wypełniamy obiekt **sb** napisem **text**.

String

```
String text = "AbCdEfAbCdEf";  
System.out.println(text);  
text.toLowerCase();  
System.out.println(text);
```

Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";  
StringBuilder sb = new StringBuilder();  
sb.append(text);  
sb.reverse();  
System.out.println(sb.toString());
```

Wypełniamy obiekt **sb** napisem **text**.

Wykonujemy operacje na obiekcie.

String

```
String text = "AbCdEfAbCdEf";  
System.out.println(text);  
text.toLowerCase();  
System.out.println(text);
```


Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";
StringBuilder sb = new StringBuilder();
sb.append(text);
sb.reverse();
System.out.println(sb.toString());
```

Wypełniamy obiekt **sb** napisem **text**.

Wykonujemy operacje na obiekcie.

Zawartość została zmieniona.

Wyświetli: **fEdCbAfEdCbA**

String

```
String text = "AbCdEfAbCdEf";
System.out.println(text);
text.toLowerCase();
System.out.println(text);
```


Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";  
StringBuilder sb = new StringBuilder();  
sb.append(text);  
sb.reverse();  
System.out.println(sb.toString());
```

Wypełniamy obiekt **sb** napisem **text**.

Wykonujemy operacje na obiekcie.

Zawartość została zmieniona.

Wyświetli: **fEdCbAfEdCbA**

String

```
String text = "AbCdEfAbCdEf";  
System.out.println(text);  
text.toLowerCase();  
System.out.println(text);
```

Wykonujemy operacje na obiekcie.

Tworzenie napisów

Przykład obrazujący różnice między napisami niemodyfikowalnymi a modyfikowalnymi.

StringBuilder

```
String text = "AbCdEfAbCdEf";
StringBuilder sb = new StringBuilder();
sb.append(text);
sb.reverse();
System.out.println(sb.toString());
```

Wypełniamy obiekt **sb** napisem **text**.

Wykonujemy operacje na obiekcie.

Zawartość została zmieniona.

Wyświetli: **fEdCbAfEdCbA**

String

```
String text = "AbCdEfAbCdEf";
System.out.println(text);
text.toLowerCase();
System.out.println(text);
```

Wykonujemy operacje na obiekcie.

Zawartość pozostała niezmieniona.

Wyświetli: **AbCdEfAbCdEf**

Podział napisów

Metoda `split()`

Aby wyodrębnić z napisu jego część, możemy wykorzystać metodę `split()` klasy `String`:

```
String string = "09-100";  
String[] parts = string.split("-");  
String part1 = parts[0];  
String part2 = parts[1];
```

Dla tej metody podajemy argument, który jest wyrażeniem regularnym.

Za ich pomocą możemy opisywać wzorce wyszukiwania bardzo skomplikowanych łańcuchów znaków.

Wyrażenia regularne będziemy omawiać w kolejnych etapach kursu.

Możemy dokonać podziału wpisując pojedynczy znak lub słowo, np.:

```
String[] parts = string.split("tak");  
String[] parts = string.split("o");
```

Podział napisów

Metoda `split()`

Aby wyodrębnić z napisu jego część, możemy wykorzystać metodę `split()` klasy `String`:

```
String string = "09-100";  
String[] parts = string.split("-");  
String part1 = parts[0];  
String part2 = parts[1];
```

"-" – wzorzec podziału.

Dla tej metody podajemy argument, który jest wyrażeniem regularnym.

Za ich pomocą możemy opisywać wzorce wyszukiwania bardzo skomplikowanych łańcuchów znaków.

Wyrażenia regularne będziemy omawiać w kolejnych etapach kursu.

Możemy dokonać podziału wpisując pojedynczy znak lub słowo, np.:

```
String[] parts = string.split("tak");  
String[] parts = string.split("o");
```

Podział napisów

Metoda `split()`

Aby wyodrębnić z napisu jego część, możemy wykorzystać metodę `split()` klasy `String`:

```
String string = "09-100";  
String[] parts = string.split("-");  
String part1 = parts[0];  
String part2 = parts[1];
```

"-" – wzorzec podziału.

`parts[0]`; `parts[1]`; – elementy powstałe z podziału.

Dla tej metody podajemy argument, który jest wyrażeniem regularnym.

Za ich pomocą możemy opisywać wzorce wyszukiwania bardzo skomplikowanych łańcuchów znaków.

Wyrażenia regularne będziemy omawiać w kolejnych etapach kursu.

Możemy dokonać podziału wpisując pojedynczy znak lub słowo, np.:

```
String[] parts = string.split("tak");  
String[] parts = string.split("o");
```

Podział napisów

Klasa StringTokenizer

Możemy również użyć metod klasy **StringTokenizer** należącej do pakietu **java.util**.

Domyślnie StringTokenizer dokona podziału korzystając z poniższego zestawu znaków:

`"\t", "\n", "\r", "\f"`

Przykład:

```
String text = "Tekst do podziału";  
StringTokenizer sToken = new StringTokenizer(text);
```

Powyższy przykład tworzenia obiektu klasy **StringTokenizer** dokona podziału napisu **text** wg znaku spacji.

Podział napisów

Klasa StringTokenizer

Możemy również sami zdefiniować własny znak podziału podając go jako drugi parametr:

```
StringTokenizer(String str, String delimiter);  
StringTokenizer strToken = new StringTokenizer(str, ",");
```

Możemy również podać kilka znaków podziału:

```
StringTokenizer strToken = new StringTokenizer(str, ", . ");
```

Podział napisów

Klasa StringTokenizer

Możemy również sami zdefiniować własny znak podziału podając go jako drugi parametr:

```
StringTokenizer(str, String delimiter);  
StringTokenizer strToken = new StringTokenizer(str, ",");
```

Możemy również podać kilka znaków podziału:

```
StringTokenizer strToken = new StringTokenizer(str, ",. ");
```

delimiter – podajemy znaki podziału,

Podział napisów

Klasa StringTokenizer

Możemy również sami zdefiniować własny znak podziału podając go jako drugi parametr:

```
StringTokenizer(String str, String delimiter);  
StringTokenizer strToken = new StringTokenizer(str, ",");
```

Możemy również podać kilka znaków podziału:

```
StringTokenizer strToken = new StringTokenizer(str, ",. ");
```

delimiter – podajemy znaki podziału,

"," – znakiem podziału będzie przecinek,

Podział napisów

Klasa StringTokenizer

Możemy również sami zdefiniować własny znak podziału podając go jako drugi parametr:

```
StringTokenizer(String str, String delimiter);  
StringTokenizer strToken = new StringTokenizer(str, ",");
```

Możemy również podać kilka znaków podziału:

```
StringTokenizer strToken = new StringTokenizer(str, ", . ");
```

delimiter – podajemy znaki podziału,

"," – znakiem podziału będzie przecinek,

"," . " – znakiem podziału będzie przecinek, kropka oraz spacja.

Podział napisów

Klasa StringTokenizer

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
strToken.countTokens();  
strToken.nextToken();
```

Podział napisów

Klasa StringTokenizer

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
strToken.countTokens();  
strToken.nextToken();
```

Przy pomocy metody **countTokens()** możemy ustalić liczbę elementów powstałych po podziale.

Podział napisów

Klasa StringTokenizer

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
strToken.countTokens();  
strToken.nextToken();
```

Przy pomocy metody **countTokens()** możemy ustalić liczbę elementów powstałych po podziale.

Pobranie kolejnych elementów podziału możemy uzyskać wywołując metodę **nextToken()**.

Podział napisów

Klasa StringTokenizer

Gdy wywołujemy **nextToken()**, a nie ma już następnych elementów, otrzymamy wyjątek:

```
Exception in thread "main"  
java.util.NoSuchElementException
```

Dlatego pobieranie kolejnych elementów podziału najlepiej wykonać z wykorzystaniem pętli **while** oraz metody **hasMoreTokens()**:

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
while (strToken.hasMoreTokens()) {  
    String s = strToken.nextToken();  
}
```

Podział napisów

Klasa StringTokenizer

Gdy wywołujemy **nextToken()**, a nie ma już następnych elementów, otrzymamy wyjątek:

```
Exception in thread "main"  
java.util.NoSuchElementException
```

Dlatego pobieranie kolejnych elementów podziału najlepiej wykonać z wykorzystaniem pętli **while** oraz metody **hasMoreTokens()**:

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
while (strToken.hasMoreTokens()) {  
    String s = strToken.nextToken();  
}
```

strToken.hasMoreTokens() – zwróci **true** jeżeli są jeszcze elementy,

Podział napisów

Klasa StringTokenizer

Gdy wywołujemy **nextToken()**, a nie ma już następnych elementów, otrzymamy wyjątek:

```
Exception in thread "main"  
java.util.NoSuchElementException
```

Dlatego pobieranie kolejnych elementów podziału najlepiej wykonać z wykorzystaniem pętli **while** oraz metody **hasMoreTokens()**:

```
String str = "To jest tekst do podziału";  
StringTokenizer strToken = new StringTokenizer(str);  
while (strToken.hasMoreTokens()) {  
    String s = strToken.nextToken();  
}
```

strToken.hasMoreTokens() – zwróci **true** jeżeli są jeszcze elementy,

strToken.nextToken() – pobieramy kolejny element.

Zadania

Wykonaj zadania z działu

Podział napisów