Zapis / odczyt plików tekstowych

Czego się dowiesz

- Jak tworzyć pliki i foldery za pomocą kodu Javy,
- w jaki sposób odczytywać i zapisywać informacje tekstowe z i do plików,
- jak wykorzystywać
  klasy File, FileWriter, FileReader, BufferedWriter i BufferedReader.

Wstęp

Programy, które tworzymy od początku kursu stają się coraz ciekawsze i mają coraz to więcej możliwości. Ciągle problemem pozostaje to, że dane odbierane od użytkownika są nietrwałe. Po zakończeniu działania programu wszelkie wprowadzone informacje zostają utracone.

Java jak w zasadzie większość języków programowania udostępnia wygodny sposób zapisu i odczytu danych z plików. Poniżej wymienione są najważniejsze klasy, które w tym pomagają:

- File - podstawowa klasa, która pozwala przechowywać informacje o pliku, a także o katalogach, ale nie służy bezpośrednio do operacji na nich,
- FileReader i FileWriter - klasy stosunkowo niskopoziomowe. Dają podstawowe możliwości zapisu i odczytu plików tekstowych znak po znaku,
- BufferedWriter i BufferedReader - klasy, które pozwalają obudować różnego typu strumienie - np. *FileReader* i *FileWriter*, dzięki czemu zapis lub odczyt są buforowane a dzięki temu wydajniejsze,
- PrintWriter - klasa, która jest podobna do PrintStream (czyli ta, którą wykorzystujemy do System.out.print()) - pozwala na prosty zapis danych do pliku tekstowego. W odróżnieniu od BufferedWritera nie wykorzystuje bufora, więc raczej powinno się ją wykorzystywać do zapisu niewielkiej ilości danych,

- Scanner - można go wykorzystać do odczytu plików podobnie jak do odczytu danych z konsoli.

Wszystkie otwarte strumienie zapisu i odczytu powinny być zamykane za pomocą metody close(). Zwolni ona używane zasoby komputera (pamięć), a także zapewni, że wszystkie informacje znajdujące się w buforze zostały zapisane do pliku, jeśli o tym zapomnisz, część danych może zostać utracona.

Tworzenie plików i folderów

W Javie w bardzo prosty sposób możemy utworzyć nowy plik lub folder, wykorzystując klasę File. Poniższy kod tworzy nowy obiekt typu File, sprawdza, czy plik o wskazanej nazwie istnieje, a jeśli nie to go tworzy.

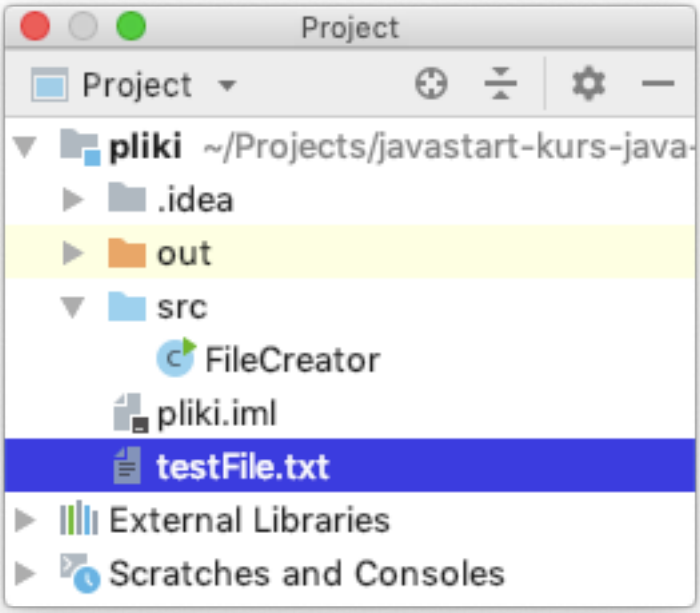
FileCreator.java

```
import java.io.File;
import java.io.IOException;

class FileCreator {
    public static void main(String[] args) {
        String fileName = "testFile.txt";
        File file = new File(fileName);

        boolean fileExists = file.exists();
        if (!fileExists) {
            try {
                fileExists = file.createNewFile();
            } catch (IOException e) {
                System.out.println("Nie udało się utworzyć pliku");
            }
        }

        if (fileExists)
            System.out.println("Plik " + fileName + " istnieje lub został utworzony");
    }
}
```

Dobrą praktyką jest nie wykorzystywanie nazw plików jako napisów bezpośrednio w konstruktorze, ale wykorzystywanie do tego zmiennych, czy stałych - dzięki temu unikamy ryzyka popełnienia literówek, jeśli odwołujemy się do jednej nazwy pliku w wielu miejscach kodu. Utworzyliśmy więc zmienną fileName typu String i zainicjowaliśmy ją nazwą pliku, który chcemy utworzyć. Możemy w tym miejscu podać samą nazwę pliku, wtedy zostanie on utworzony w katalogu naszego projektu, albo ścieżkę absolutną typu *C:/a/b/c.txt*.

Obiekt klasy File o nazwie file jest reprezentacją pliku lub folderu w naszym programie. Możemy na nim wykonać pewne operacje, np. możemy sprawdzić, czy taki plik rzeczywiście istnieje na komputerze. Wykorzystujemy do tego metodę exists(), która zwraca *true* lub *false* w zależności od tego, czy plik lub folder istnieje, czy nie. Jeśli plik nie istnieje, to tworzymy go za pomocą metody createNewFile(), która może generować wyjątek kontrolowany IOException. Jeśli faktycznie taka sytuacja by wystąpiła, wyświetlamy komunikat o niepowodzeniu w bloku catch, jeśli wszystko przebiegnie bez zakłóceń wyświetlamy potwierdzenie. Wyjątek może wystąpić np. w sytuacji, gdy nie mamy praw zapisu.

W IntelliJ plik pojawi się od razu w strukturze projektu.



W eclipse wciśnij klawisz *F5* lub kliknij prawym przyciskiem myszy na nazwę projektu i wybierzesz opcję *Refresh.*

W podobny sposób jak powyżej można utworzyć nowy folder, jednak służy do tego metoda mkdir(). Jeżeli utworzony ma zostać ciąg folderów, np "/a/b/c" skorzystaj z metody mkdirs(). Jako uzupełnienie warto pamiętać, że tworząc foldery możemy wskazywać ścieżki relatywne lub absolutne:

D:/a - utworzony zostanie folder na dysku D o nazwie "a"
/a - utworzony zostanie katalog dysku systemowym
./a - utworzony zostanie folder w katalogu projektu
../a - utworzony zostanie folder w folderze powyżej folderu projektu (w przypadku uruchamiania programu eclipse, bedzie to „workspace")

Odczyt plików - Scanner

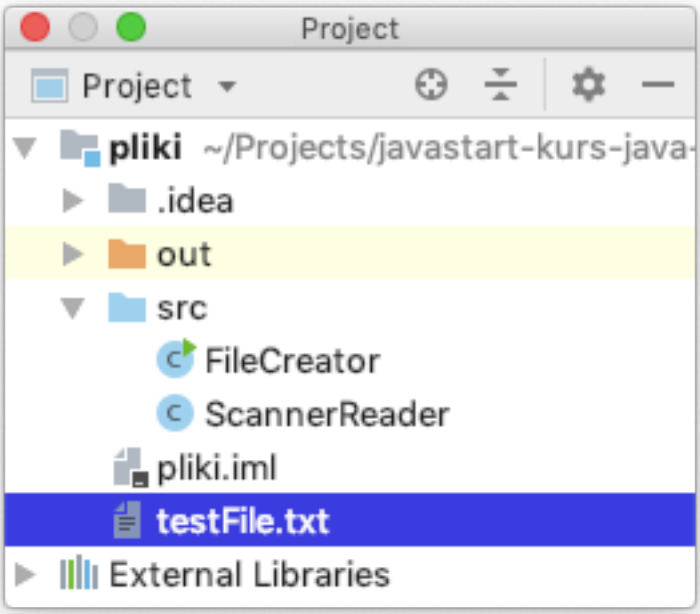
Najprostszym sposobem odczytu plików tekstowych jest skorzystanie ze znanej nam już klasy Scanner. Jej zachowanie jest identyczne jak w przypadku odczytu z konsoli, zmienia się jedynie źródło z którego tego odczytu dokonujemy.

W przypadku odczytu z plików najczęściej będziemy korzystali z połączenia dwóch metod:

boolean hasNextLine() - sprawdza, czy w pliku jest jeszcze kolejny wiersz do odczytania,
String nextLine() - wczytuje i zwraca kolejny wiersz z pliku.

Jeżeli w pliku mamy zapisany np. ciąg liczb to analogicznie moglibyśmy używać metod typu hasNextInt() i nextInt(), przy czym metoda hasNextInt() traktuje jako separator pomiędzy liczbami dowolny biały znak, np. spację, albo tabulator, a nie tylko enter.

Dlaczego akurat taka kombinacja metod? Odczytując dane z pliku nie wiemy ile znaków, czy ile wierszy tekstu się w nim znajduje, dopóki tego pliku nie odczytamy "z góry na dół". Metoda hasNextLine() pozwoli nam więc sprawdzić warunek "czy w pliku jest jeszcze coś do wczytania" a metoda nextLine() wczyta te dane jeśli istnieją. Kombinacja ta świetnie sprawdza się w połączeniu z pętlą while. Spójrzmy na przykład. W głównym katalogu projektu (nie src) stwórz plik testFile.txt z dowolną zawartością.



testFile.txt

```
Ania
Kasia
Basia
```

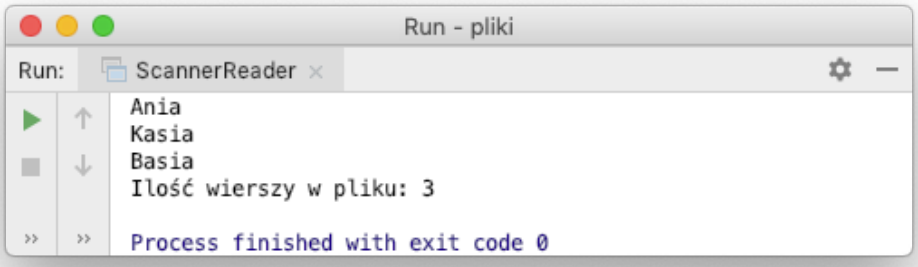
W programie chcemy odczytać i wyświetlić kolejne imiona oraz zliczyć ilość wierszy w pliku.

ScannerReader.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class ScannerReader {
    public static void main(String[] args) throws FileNotFoundException
{
        String fileName = "testFile.txt";
        File file = new File(fileName);
        Scanner scan = new Scanner(file);

        int lines = 0;
        while (scan.hasNextLine()) {
            String name = scan.nextLine();
            System.out.println(name);
            lines++;
        }
        System.out.println("Ilość wierszy w pliku: " + lines);
        scan.close();
    }
}
```

Tworzymy zmienną przechowującą nazwę pliku oraz obiekt typu File reprezentujący ten plik. Dalej utworzyliśmy Scanner, jednak skorzystaliśmy z konstruktora, który przyjmuje obiekt File, a nie tak jak do tej pory obiekt Sytem.in. Dalej deklarujemy zmienną lines, która posłuży nam do zliczenia ilości wierszy w pliku. Konstruktor przyjmujący jako argument obiekt typu File deklaruje kontrolowany wyjątek FileNotFoundException, który dla czytelności kodu zadeklarowaliśmy w metodzie main() poprzez throws, jednak w realnym programie dobrze byłoby użyć w tym miejscu bloku try-catch i poinformować użytkownika o błędzie.

Pętlę while możemy przetłumaczyć na nasz język jako "dopóki z pliku jest jeszcze kolejny wiersz tekstu scan.hasNextLine() to go wczytaj i wyświetl oraz zwiększ zmienną lines o 1". Kiedy dojdziemy do ostatniego wiersza pliku, metoda hasNextLine() zwróci false, a więc pętla się zakończy. Wyświetlamy wtedy wartość zmiennej lines. Pracując ze strumieniami, czyli np. odczytem z plików powinniśmy pamiętać o ich zamykaniu. O ile przy odczycie nie niesie to ogromnych konsekwencji, to zobaczymy, że przy zapisie pojawią się problemy jeśli o tym zapomnimy.



Odczyt plików - BufferedReader

Odczyt plików z wykorzystaniem Scannera jest bardzo wygodny jednak posiada jedną wadę - średnią wydajność. Najwolniejsze fragmenty niemal każdej aplikacji to te związane z operacjami wejścia-wyjścia, takimi jak odczyt/zapis plików,

czy też komunikacja z innym urządzeniem poprzez internet. W przypadku małych plików nie jest to duży problem, jednak w przypadku dużych plików problem narasta. Scanner ma z góry narzucony bufor wielkości 1024 bajtów, natomiast istnieje klasa BufferedReader o domyślnym buforze 8192 bajtów, który dodatkowo możemy zmienić (np. jeszcze bardziej powiększyć). Dzięki większemu buforowi, do pamięci komputera jednorazowo będzie wczytany większy fragment pliku, aplikacja będzie się dzięki temu rzadziej komunikowała z dyskiem twardym, a tym samym będzie szybciej działała.

BufferedReader wymaga jednak do działania innego obiektu klasy dziedziczącej po klasie Reader. Do odczytu plików taką klasą jest FileReader. Różnica pomiędzy FileReader, a BufferedReader jest taka, że ta pierwsza pozwala nam czytać plik znak po znaku, natomiast ta druga wiersz po wierszu.

Zobaczmy więc jak wcześniejszy przykład wyglądałby z wykorzystaniem klasy BufferedReader, a nie Scanner.

FileTester.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class FileTester {
    public static void main(String[] args) {
        String fileName = "testFile.txt";
        FileReader fileReader = null;
        BufferedReader reader = null;
        try {
            fileReader = new FileReader(fileName);
            reader = new BufferedReader(fileReader);
            String nextLine = null;
            int lines = 0;
            while ((nextLine = reader.readLine()) != null) {
                System.out.println(nextLine);
                lines++;
            }
            System.out.println("Ilość wierszy w pliku: " + lines);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (reader != null)
                    reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Utworzenie nowego obiektu BufferedReader jest dosyć długie - klasa FileReader pozwala pracować na obiekcie File, a BufferedReader dodaje do obiektu FileReader dodatkowe metody - jest to wzorzec projektowy zwany dekoratorem.

Kolejne wiersze możemy odczytać za pomocą metody readLine() i przypisać je tak jak w naszym przykładzie do tymczasowej zmiennej nextLine typu *String*.
Jeśli readLine() zwróci wartość null, oznacza to, że doszliśmy do końca pliku i należy wyjść z pętli.

W przypadku odczytu pliku wyjątki mogą wystąpić przy tworzeniu obiektu FileReader (*FileNotFoundException*) oraz przy nieoczekiwanym problemie odczytu pliku (IOException). Ponieważ wyjątek FileNotFoundException dziedziczy po IOException, możemy je obsłużyć w jednym ogólnym bloku catch.

Podobnie jak w przypadku Scannera zamykamy strumień, z którego korzystaliśmy. Najczęściej wykorzystywany będzie do tego blok finally w formie jaką przedstawioną, jednak w Javie 7 da się to zrobić ładniej o czym dowiesz się poniżej.

Po uruchomieniu w konsoli powinien pojawić się identyczny wydruk jak poprzednio.

try with resources

Jeśli porównamy kod do odczytu plików z wykorzystaniem klas Scanner oraz BufferedReader to nie da się ukryć, że ten drugi jest dużo bardziej skomplikowany, choć robi w zasadzie to samo. Szczególnie negatywnie na czytelność wpływają tutaj zagnieżdżone bloki try-catch, rozdzielona deklaracja i inicjalizacja obiektów FileReader i BufferedReader. Projektanci Javy zauważyli ten problem i rozwiązali go wprowadzając w Javie 7 konstrukcję nazywaną try-with-resources. Jest ona przeznaczona do pracy z klasami służącymi do operacji na zasobach. Jeśli taka klasa implementuje interfejs AutoCloseable, to

będziemy zwolnieni z konieczności jawnego wywoływania metody close(), zostanie to za nas zrobione automatycznie po wykonaniu się bloku try. Spójrzmy jak może wyglądać poprzedni przykład z wykorzystaniem tej konstrukcji.

FileTesterTry.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class FileTesterTry {
    public static void main(String[] args) {
        String fileName = "testFile.txt";

        try (
                var fileReader = new FileReader(fileName);
                var reader = new BufferedReader(fileReader);
        ) {
            String nextLine = null;
            int lines = 0;
            while ((nextLine = reader.readLine()) != null) {
                System.out.println(nextLine);
                lines++;
            }
            System.out.println("Ilość wierszy w pliku: " + lines);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Deklarację i inicjalizację obiektów FileReader i BufferedReader przenieśliśmy między nawiasy okrągłe występujące bezpośrednio po słowie try. Wszystkie obiekty, które zostaną utworzone w tym miejscu, a implementują interfejs AutoCloseable (w naszym przypadku oba z nich) zostaną automatycznie zamknięte, więc możemy pozbyć się także mało czytelnego bloku finally. Funkcjonalnie kod jest identyczny z tym, który stworzyliśmy wcześniej, jednak jego czytelność jest bez porównania lepsza. Od Javy 10 możemy dodatkowo użyć słowa var, aby jeszcze skrócić zapis.

Całość dałoby się jeszcze bardziej skrócić sprowadzając tworzenie BufferedReadera do jednej linijki:

```
var reader = new BufferedReader(new FileReader("testFile.txt"));
```

Należy ocenić co jest jednak dla nas czytelniejsze.

W Javie 9 wprowadzono kolejne usprawnienie bloku try-with-resoures, które polega na tym, że deklaracja i inicjalizacja obiektów, które mają być zamknięte nie musi odbywać się bezpośrednio w okrągłych nawiasach przy try. Teraz deklaracja może znajdować się przed blokiem try, a obiekty, które mają być zamknięte należy wymienić w okrągłych nawiasach. Jedyne ograniczenie jest takie, że zmienne muszą być finalne lub efektywnie finalne, czyli nie możemy później przypisywać do nich nowego obiektu.

Wcześniejszy przykład z zastosowaniem tego mechanizmu wyglądałby więc np. tak:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileTesterTry {
    public static void main(String[] args) throws IOException {
        String fileName = "testFile.txt";
        var fileReader = new FileReader(fileName);
        var reader = new BufferedReader(fileReader);

        try (
                fileReader;
                reader;
        ) {
            String nextLine = null;
            int lines = 0;
            while ((nextLine = reader.readLine()) != null) {
                System.out.println(nextLine);
                lines++;
            }
            System.out.println("Ilość wierszy w pliku: " + lines);
        }

    }
}
```

Przy okazji widać, że dzięki konstrukcji try-with-resources możemy zapisać blok try, z którym nie jest powiązany żaden catch, ani sekcja finally (podobnie działa to także w Javie 7 i 8, jednak deklaracja obiektów musiała znajdować się w okrągłych nawiasach).

Zapis plików

Zapis do plików jest równie prosty i podobny co ich odczyt. Najlepiej jest do tego wykorzystać kombinację obiektów Buffered**Writer** i File**Writer***.* Widać tutaj silną analogię do przed chwilą wykorzystywanych klas BufferedReader i FileReader.

FileWriter pozwala zapisywać do pliku dane znak po znaku, a *BufferedWriter* wiersz po wierszu.

Stwórzmy klasę, w której zapiszemy do pliku *testFile.txt* kilka imion jedno pod drugim.

FileWriterTest.java

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

class FileWriterTest {
    public static void main(String[] args) {
        String fileName = "testFile.txt";
        try (
                var fileWriter = new FileWriter(fileName);
                var writer = new BufferedWriter(fileWriter);
        ) {
            writer.write("Bolek");
            writer.newLine();
            writer.write("Lolek");
            writer.newLine();
            writer.write("Karolek");
        } catch (IOException e) {
            System.err.println("Nie udało się zapisać pliku " + fileName);
        }
    }
}
```

Proces tworzenia BufferedWritera przebiega analogicznie jak w przypadku BufferedReadera i również dałoby się go skrócić do jednego wiersza.

Do zapisu danych w pliku wykorzystujemy metodę write(), która jako argument przyjmuje dowolny String. Istotne jest to, żeby do zapisywania znaku nowej linii używać dedykowanej metody newLine(), dzięki czemu mamy gwarancję, że zostanie wykorzystany znak nowego wiersza odpowiedni dla danego systemu operacyjnego.

Ponieważ korzystamy z konstrukcji try-with-resources to nie musimy pamiętać o wywołaniu metody close(), jednak jeżeli korzystalibyśmy ze standardowego try-catch i zapomnimy wywołać close() to część danych, która jest w buforze, nie zostanie zapisana. Należy na to zwrócić szczególną uwagę i przynajmniej wywołać metodę flush(), która opróżnia bufor.

Jeśli uruchomimy program i zajrzymy do pliku *testFile.txt* to zauważymy, że znajdują się w nim trzy wiersze, które zdefiniowaliśmy w aplikacji.

```
Bolek
Lolek
Karolek
```

Problem w tym, że straciliśmy dane, które były w nim zapisane wcześniej. Na szczęście łatwo możemy naprawić ten problem i jeśli chcemy dopisywać dane do pliku, a nie zapisywać plik od zera, to wystarczy dodać flagę true w konstruktorze FileWritera.

```
var fileWriter = new FileWriter(fileName, true);
```

Przy każdym uruchomieniu tej klasy w pliku będą pojawiały się kolejne wiersze tekstu pozostawiając te już istniejące.