# Notatki - Internationalization i Localization

## Spis treści



## Internationalization i Localization

Na początek pogadamy sobie o lokalizacji. Dużo aplikacji jest używanych przez użytkowników z różnych krajów, a co za tym idzie, aplikacje muszą funkcjonować w różnych językach. Oprócz tego, że aplikacje powinny funkcjonować z możliwością wyboru języka, to często konieczne jest zapisywanie dat lub pieniędzy w określonych formatach. Mogą to być również przecinki lub kropki w liczbach.

Przykład, nad którym warto się chwilę zastanowić. Data `6/1/10` dla nas oznacza `6 styczeń 2010`, w US natomiast będzie oznaczała `1 czerwiec 2010`. Jak już o tym wiemy, to pisząc program należy takie sytuacje wspierać i obsługiwać 😉.

Wyjaśnijmy sobie dwa terminy:

- **Internationalization** - proces projektowania programu w taki sposób, aby można było ten program dostosowywać do różnych języków lub regionów bez dokonywania zmian w kodzie. Oznacza to, że aplikacja, którą piszemy może być konfigurowalna przez pliki z "jakimiś ustawieniami". Efektem jest sytuacja, w której w zależności od ustawień możemy stosować inne formaty dat dla innych preferencji użytkownika. W zasadzie, to nie musimy wspierać wielu języków jednocześnie, wystarczy wspierać więcej niż jeden język. Termin ten odnosi się do tego, że jesteś przygotowany na obsługę wielu języków w swoim oprogramowaniu.
- **Localization** - proces dostosowywania oprogramowania do określonego regionu lub języka. W wyniku takiego procesu, aplikacja faktycznie wspiera wiele lokalizacji (dalej będziemy używać określenia `locale`). Java określa lokalizację (`locale`) jako konkretny region geograficzny/polityczny/kulturowy. `Locale` można rozumieć jako para język oraz kraj. Możliwa jest sytuacja gdzie kilka krajów używa tego samego języka, ale w zależności od kraju, różnią się pewne aspekty formatowania. Mówiąc lokalizacja, mamy też na myśli, że aplikacja wspiera słowa/zdania/stringi w różnych językach. Jednocześnie zawiera w tym się kwestia formatowania dat i liczb w formacie zgodnym z danym `locale`. Za każdym razem jak będziemy dodawać do aplikacji nowy język, który ma być wspierany, musimy mieć wsparcie dla wszystkich istniejących tekstów/formatów wiążących się z danym językiem.

Jedno i drugie słowo jest dosyć długie, a jak już może udało Ci się zauważyć, programersy lubią skracać słowa/zdania/czynności. Dlatego też, powstały 2 skróty dla powyższych słów:

- **i18n** - Internationalization
- **l10n** - Localization

Liczba po drodze odnosi się do liczby znaków pomiędzy znakami skrajnymi.

Przechodząc do konkretów.

# Podstawy Locale

## Jak zdefiniować default Locale

```
Locale defaultLocale = Locale.getDefault();
System.out.println(defaultLocale);
```

Na ekranie zostanie wydrukowane `pl_PL`, gdzie:

- **pl** - oznacza, że na komputerze jest ustawiony język polski
- **PL** - oznacza, że ustawiony kraj na komputerze to Polska

Czyli pierwszy fragment przed podłogą oznacza język, a drugi po podłodze oznacza kraj. Kraj nie jest konieczny, Locale może równie dobrze wyglądać tak `pl`.

## A jak można samemu określić Locale

```
System.out.println(Locale.ENGLISH); // en
System.out.println(Locale.UK); // en_GB
```

Albo przez konstruktor:

```
System.out.println(new Locale("en")); // en
System.out.println(new Locale("en", "PL")); // en_PL - no tak też można
```

Default można też zmienić na pojedyncze uruchomienie programu, przy ponownym uruchomieniu zostaną wczytane poprzednie ustawienia.

Powyższy przykład będzie działał poprawnie, gdy korzystamy z wersji Java mniejszej niż 19. Od Java 19 zaleca się wykorzystanie metody `Locale.of()`.

# Notatki - Resource Bundle

## Spis treści



## Resource Bundle

Wspomnieliśmy o **i18n** oraz **l10n** i pojawia się pytanie, po co nam to? Pokażmy przykład użycia tego w praktyce - **Resource Bundle** ☺.

`Resource Bundle` służy do zapisu konkretnych ustawień aplikacji i użycia ich w zależności od podanego `Locale`. Resource Bundle może być przedstawione za pomocą plików zwanych `property file` albo ustawienia mogą też być definiowane w plikach `.java`.

Na co dzień zamiast `property file` używa się stwierdzenia **plik z propertasami** ☺.

Plik taki jest zdefiniowany na zasadzie Mapy/Słownika - `klucz=wartość`

```
property1=someValue1
property2=someValue2
```

Dotychczas, jeżeli pisalibyśmy jakiś program, który używa Stringów, to Stringi były pisane od razu w kodzie po polsku. Teraz dowiemy się w jaki sposób można napisać aplikację, gdzie Stringi mogą zostać wyciągnięte "poza" aplikację, żeby mieć je w oddzielnym pliku i jak będziemy potrzebowali dodać kolejny język do aplikacji to dodajemy kolejny plik z językiem. Ale przechodząc do konkretów.

## Sklep Internetowy!

Przejdźmy do przykładu sklepu internetowego, który ma działać i w języku polskim i angielskim.

Tutaj chcę jeszcze dodać, że nie będziemy się zagłębiać w różne sposoby zapisu danych w plikach `.properties`. Dodam tylko, że można tam pisać komentarze zaczynając linijkę od `#` lub `!`

Pliki `.properties` nie są jedynym możliwym sposobem na zdefiniowanie ustawień/tłumaczeń. Możliwe jest to również do osiągnięcia w samych plikach `.java`. Nie pokazujemy jednak tego sposobu, ze względu na raczej mało popularne jego zastosowanie.

Przykładowe pliki `.properties`:

*Plik Store.properties*

```
defaultProperty=Default Value
```

*Plik Store_en.properties*

```
mainPage=Main Page!
footer=And here we have our footer
```

*Plik Store_pl.properties*

```
mainPage=Strona Główna!
footer=A oto jest stopka
```

# Wartość domyślna

Mając już określone pliki z ustawieniami jak w przykładzie powyżej, możemy przejść do napisania kodu, który odczyta te wartości z plików. W tym celu wykorzystamy klasę `ResourceBundle`:

```
public class InternetStoreExample {

    public static void main(String[] args) {
        printValuesForDefault();
    }

    private static void printValuesForDefault() {
        System.out.println("locale: " + Locale.getDefault());

        ResourceBundle resourceBundle = ResourceBundle.getBundle("Store");
        String mainPage = resourceBundle.getString("mainPage");
        String footer = resourceBundle.getString("footer");

        System.out.println("mainPage: " + mainPage);
        System.out.println("footer: " + footer);
    }

}
```

W przykładzie powyżej drukowane wartości na ekranie będą pobrane z pliku `.properties` odpowiedniego do ustawionego domyślnego `Locale`. Można to sprawdzić wywołując: `Locale.getDefault()`.

# Konkretne wartości dla danego Locale

Istnieje również możliwość pobrania wartości z pliku dla konkretnego, narzuconego przez nas locale. Przykład poniżej.

```
public class InternetStoreExample {

    public static void main(String[] args) {
        Locale pl = new Locale("pl", "PL");
        Locale en = Locale.UK;

        printValues(pl);
        printValues(en);
    }

    private static void printValues(Locale locale) {
        System.out.println("locale: " + locale);

        ResourceBundle resourceBundle = ResourceBundle.getBundle("Store", locale);
        String mainPage = resourceBundle.getString("mainPage");
        String footer = resourceBundle.getString("footer");

        System.out.println("mainPage: " + mainPage);
        System.out.println("footer: " + footer);
    }

}
```

# Wydrukowanie wszystkich wartości

Poniżej znajdziesz fragment kodu, w którym drukowane są wszystkie wartości dla podanego locale.

```
public class InternetStoreExample {

    public static void main(String[] args) {
        Locale pl = new Locale("pl", "PL");
        Locale en = Locale.UK;

        printWholeMap(pl);
        printWholeMap(en);
    }

    private static void printWholeMap(Locale locale) {
        System.out.println("locale: " + locale);

        ResourceBundle resourceBundle = ResourceBundle.getBundle("Store", locale);

        Map<String, String> withValues = resourceBundle.keySet().stream()
            .collect(Collectors.toMap(key -> key, key -> resourceBundle.getString(key)));
        System.out.println(withValues);
    }

}
```

Zauważ, że mimo, że w plikach `Store_en.properties` ani `Store_pl.properties` nie ma klucza, `defaultProperty`, a został on wydrukowany na ekranie. W tym mechanizmie działa swojego rodzaju "dziedziczenie", dlatego klucz, który nie istniał w plikach `Store_en.properties` ani `Store_pl.properties`, ale istniał w pliku `Store.properties` został wydrukowany. Możemy równie dobrze ten klucz nadpisać defininiując go np. w pliku `Store_pl.properties`.

# Hierarchia poszukiwań pliku .properties

Poniżej została umieszczona tabela, w której została rozpisana hierarchia poszukiwań odpowiedniego pliku `.properties`. W krokach dodane są również pliki `.java`, ale tak jak zostało wspomniane wcześniej, nie poruszaliśmy tematyki definiowania takich ustawień w plikach `.java`. Natomiast dla porządku, pliki `.java` również zostały umieszczone w tabeli.

| Lp. | Szukamy pliku | Co się dzieje pod spodem? |
|---|---|---|
| 1 | Store_en_US.java | Szukamy podanego pliku |
| 2 | Store_en_US.properties | Szukamy podanego pliku |
| 3 | Store_en.java | Jeżeli pliku wyżej nie ma, szukamy pliku tylko z językiem, bez kraju |
| 4 | Store_en.properties | Jeżeli pliku wyżej nie ma, szukamy pliku tylko z językiem, bez kraju |
| 5 | Store_pl_PL.java | Jeżeli pliku wyżej nie ma, szukamy defaultowego Locale |
| 6 | Store_pl_PL.properties | Jeżeli pliku wyżej nie ma, szukamy defaultowego Locale |
| 7 | Store_pl.java | Jeżeli pliku wyżej nie ma, szukamy defaultowego Locale, ale bez kraju |
| 8 | Store_pl.properties | Jeżeli pliku wyżej nie ma, szukamy defaultowego Locale, ale bez kraju |
| 9 | Store.java | Jeżeli pliku wyżej nie ma, szukamy pliku bez Locale |
| 10 | Store.properties | Jeżeli pliku wyżej nie ma, szukamy pliku bez Locale |
| 11 | - | Jeżeli nie znaleziono żadnego pliku to zostaje wyrzucone `MissingResourceException` |

# Notatki - Formatting by Locale

## Spis treści



## Formatowanie numerów

Java dostarcza nam możliwość formatowania numerów na podstawie określonego `Locale`. Czyli w sumie nie musimy się zastanawiać jak formatować numery w określonym miejscu na świecie, gdzie ma być kropka, a gdzie przecinek - Java zrobi to za nas.

W tym celu możemy używać metod określonych w poniższej tabeli:

| Metoda | Co robi |
|---|---|
| NumberFormat.getInstance() | Ogólny formatter |
| NumberFormat.getNumberInstance() | To samo co powyżej |
| NumberFormat.getPercentInstance() | Do formatowania liczb z procentami |
| NumberFormat.getCurrencyInstance() | Służy do formatowania pieniążków |

**Przykłady wykorzystania:**

```
import java.text.NumberFormat;
import java.util.Locale;

public class FormattingNumbers {

    public static void main(String[] args) {
        int number = 1_234_567;
        Locale localePL = new Locale("pl", "PL");
        Locale localeUS = Locale.US;
        Locale localeGERMANY = Locale.GERMANY;

        System.out.println("NumberFormat.getInstance()");
        System.out.println("US: " + NumberFormat.getInstance(localeUS).format(number));
        System.out.println("PL: " + NumberFormat.getInstance(localePL).format(number));
        System.out.println("GERMANY: " + NumberFormat.getInstance(localeGERMANY).format(number));
        System.out.println();

        System.out.println("NumberFormat.getNumberInstance()");
        System.out.println("US: " + NumberFormat.getNumberInstance(localeUS).format(number));
        System.out.println("PL: " + NumberFormat.getNumberInstance(localePL).format(number));
        System.out.println("GERMANY: " + NumberFormat.getNumberInstance(localeGERMANY).format(number));
        System.out.println();

        System.out.println("NumberFormat.getCurrencyInstance()");
        System.out.println("US: " + NumberFormat.getCurrencyInstance(localeUS).format(number));
```

```
        System.out.println("PL: " + NumberFormat.getCurrencyInstance(localePL).format(number));
        System.out.println("GERMANY: " + NumberFormat.getCurrencyInstance(localeGERMANY).format(number));
        System.out.println();

        System.out.println("NumberFormat.getPercentInstance()");
        System.out.println("US: " + NumberFormat.getPercentInstance(localeUS).format(number));
        System.out.println("PL: " + NumberFormat.getPercentInstance(localePL).format(number));
        System.out.println("GERMANY: " + NumberFormat.getPercentInstance(localeGERMANY).format(number));
    }
}
```

# Formatowanie dat

Skoro już mowa o formatowaniu, to Java pozwala też na różnorakie sposoby formatowania dat i czasów. Najczęściej spotykanym formatem będzie ustandaryzowany format `ISO`, który można narzucić poprzez `DateTimeFormatter`:

```
LocalDate date = LocalDate.of(2001, Month.DECEMBER, 15);
LocalTime time = LocalTime.of(16, 38, 52);
LocalDateTime dateTime = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE)); // 2001-12-15
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME)); // 16:38:52
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)); // 2001-12-15T16:38:52
```

Poniżej zostały umieszczone fragmenty kodu pokazujące kolejny możliwy sposób działania klasy `DateTimeFormatter`. Załóżmy, że w każdym z poniższych przykładów będziemy się odnosić do zmiennych określonych poniżej. Kolejne przykłady również będą się odnosiły do zmiennych `date`, `time`, `dateTime`, `offsetDateTime` oraz `zonedDateTime` zdefiniowanych poniżej.

```
LocalDate date = LocalDate.of(2001, Month.DECEMBER, 15);
LocalTime time = LocalTime.of(16, 38, 52);
LocalDateTime dateTime = LocalDateTime.of(date, time);
OffsetDateTime offsetDateTime = OffsetDateTime.of(dateTime, ZoneOffset.ofHours(3));
ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, ZoneId.of("Poland"));
```

**Przykłady wykorzystania `ISO_FORMAT`:**

```
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE)); ①
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME)); ②
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)); ③
System.out.println(offsetDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)); ④
System.out.println(zonedDateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)); ⑤
```

① Zostanie wydrukowane: *2001-12-15*

② Zostanie wydrukowane: *16:38:52*

③ Zostanie wydrukowane: *2001-12-15T16:38:52*

④ Zostanie wydrukowane: *2001-12-15T16:38:52*

⑤ Zostanie wydrukowane: *2001-12-15T16:38:52*

Jeżeli chcemy podejść do tematu inaczej, mamy też możliwość wykorzystać poniższe kombinacje metod i parametrów:

| DateTimeFormatter | FormatStyle |
|---|---|
| DateTimeFormatter.ofLocalizedDate() | FormatStyle.FULL<br>FormatStyle.LONG<br>FormatStyle.MEDIUM<br>FormatStyle.SHORT |
| DateTimeFormatter.ofLocalizedTime() | FormatStyle.FULL<br>FormatStyle.LONG<br>FormatStyle.MEDIUM<br>FormatStyle.SHORT |
| DateTimeFormatter.ofLocalizedDateTime() | FormatStyle.FULL<br>FormatStyle.LONG<br>FormatStyle.MEDIUM<br>FormatStyle.SHORT |

Poniżej przykłady wykorzystania `ofLocalizedDate()`, `ofLocalizedTime()` oraz `ofLocalizedDateTime()`. Na potrzeby poniższych fragmentów kodu, wprowadzamy również zmienne takie jak:

**Definicje formatterów:**

```
DateTimeFormatter dateFormatterFULL = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
DateTimeFormatter dateFormatterLONG = DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);
DateTimeFormatter dateFormatterMEDIUM = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
DateTimeFormatter dateFormatterSHORT = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

DateTimeFormatter timeFormatterFULL = DateTimeFormatter.ofLocalizedTime(FormatStyle.FULL);
DateTimeFormatter timeFormatterLONG = DateTimeFormatter.ofLocalizedTime(FormatStyle.LONG);
DateTimeFormatter timeFormatterMEDIUM = DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);
DateTimeFormatter timeFormatterSHORT = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);

DateTimeFormatter dateTimeFormatterFULL = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL);
DateTimeFormatter dateTimeFormatterLONG = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
DateTimeFormatter dateTimeFormatterMEDIUM = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
DateTimeFormatter dateTimeFormatterSHORT = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
```

**Przykłady wykorzystania metody `ofLocalizedDate()`:**

```
System.out.println(date.format(dateFormatterFULL)); ①
System.out.println(date.format(dateFormatterLONG)); ②
System.out.println(date.format(dateFormatterMEDIUM)); ③
System.out.println(date.format(dateFormatterSHORT)); ④
```

① Zostanie wydrukowane: *sobota, 15 grudnia 2001*

② Zostanie wydrukowane: *15 grudnia 2001*

③ Zostanie wydrukowane: *15 gru 2001*

④ Zostanie wydrukowane: *15.12.2001*

**Przykłady wykorzystania metody `ofLocalizedTime()`:**

```
System.out.println(time.format(timeFormatterFULL)); ①
System.out.println(time.format(timeFormatterLONG)); ②
System.out.println(time.format(timeFormatterMEDIUM)); ③
System.out.println(time.format(timeFormatterSHORT)); ④
```

① Exception message: *Unable to extract ZoneId from temporal 16:38:52 with chronology ISO*

② Exception message: *Unable to extract ZoneId from temporal 16:38:52 with chronology ISO*

③ Zostanie wydrukowane: *16:38:52*

④ Zostanie wydrukowane: *16:38*

**Przykłady wykorzystania metody `ofLocalizedDateTime()` z wykorzystaniem `LocalDateTime`:**

```
System.out.println(dateTime.format(dateTimeFormatterFULL)); ①
System.out.println(dateTime.format(dateTimeFormatterLONG)); ②
System.out.println(dateTime.format(dateTimeFormatterMEDIUM)); ③
System.out.println(dateTime.format(dateTimeFormatterSHORT)); ④
```

① Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

② Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

③ Zostanie wydrukowane: *15 gru 2001, 16:38:52*

④ Zostanie wydrukowane: *15.12.2001, 16:38*

**Przykłady wykorzystania metody `ofLocalizedDateTime()` z wykorzystaniem `OffsetDateTime`:**

```
System.out.println(offsetDateTime.format(dateTimeFormatterFULL)); ①
System.out.println(offsetDateTime.format(dateTimeFormatterLONG)); ②
System.out.println(offsetDateTime.format(dateTimeFormatterMEDIUM)); ③
System.out.println(offsetDateTime.format(dateTimeFormatterSHORT)); ④
```

① Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52+03:00*

② Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52+03:00*

③ Zostanie wydrukowane: *15 gru 2001, 16:38:52*

④ Zostanie wydrukowane: *15.12.2001, 16:38*

**Przykłady wykorzystania metody `ofLocalizedDateTime()` z wykorzystaniem `ZonedDateTime`:**

```
System.out.println(zonedDateTime.format(dateTimeFormatterFULL)); ①
System.out.println(zonedDateTime.format(dateTimeFormatterLONG)); ②
System.out.println(zonedDateTime.format(dateTimeFormatterMEDIUM)); ③
System.out.println(zonedDateTime.format(dateTimeFormatterSHORT)); ④
```

① Zostanie wydrukowane: *sobota, 15 grudnia 2001 16:38:52 czas środkowoeuropejski standardowy*

② Zostanie wydrukowane: *15 grudnia 2001 16:38:52 CET*

③ Zostanie wydrukowane: *15 gru 2001, 16:38:52*

④ Zostanie wydrukowane: *15.12.2001, 16:38*

Należy tylko pamiętać, że `FormatStyle.FULL` oraz `FormatStyle.LONG` stara się pokazać również strefę czasową, zatem stosowanie ich do dat lub czasów bez podanych stref czasowych powoduje błąd podczas działania programu. Chyba, że mamy samą datę, wtedy działa to bez błędu.

No dobrze, ale nigdzie jeszcze nie zostało wspomniane o połączeniu `DateTimeFormatter` z `Locale`. Już podajemy przykład.

```
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM).withLocale(Locale.US);
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT).withLocale(Locale.GERMAN);
```

Jak widzisz, po określeniu `DateTimeFormatter` należy jeszcze dodać `.withLocale()`, inaczej będziemy używać pod spodem `Locale.getDefault()`.

Poniżej znajdziesz więcej przykładów:

**Przykłady wykorzystania `Locale` dla `LocalDateTime`:**

```
System.out.println("GRM: " + localDateTime.format(dateTimeFormatterFULL.withLocale(Locale.GERMAN))); ①
System.out.println("US: " + localDateTime.format(dateTimeFormatterFULL.withLocale(Locale.US))); ②

System.out.println("GRM: " + localDateTime.format(dateTimeFormatterLONG.withLocale(Locale.GERMAN))); ③
System.out.println("US: " + localDateTime.format(dateTimeFormatterLONG.withLocale(Locale.US))); ④

System.out.println("GRM: " + localDateTime.format(dateTimeFormatterMEDIUM.withLocale(Locale.GERMAN))); ⑤
System.out.println("US: " + localDateTime.format(dateTimeFormatterMEDIUM.withLocale(Locale.US))); ⑥

System.out.println("GRM: " + localDateTime.format(dateTimeFormatterSHORT.withLocale(Locale.GERMAN))); ⑦
System.out.println("US: " + localDateTime.format(dateTimeFormatterSHORT.withLocale(Locale.US))); ⑧
```

① Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

② Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

③ Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

④ Exception message: *Unable to extract ZoneId from temporal 2001-12-15T16:38:52*

⑤ Zostanie wydrukowane: *GRM: 15.12.2001, 16:38:52*

⑥ Zostanie wydrukowane: *US: Dec 15, 2001, 4:38:52 PM*

⑦ Zostanie wydrukowane: *GRM: 15.12.01, 16:38*

⑧ Zostanie wydrukowane: *US: 12/15/01, 4:38 PM*

**Przykłady wykorzystania `Locale` dla `ZonedDateTime`:**

```
System.out.println("GRM: " + zonedDateTime.format(dateTimeFormatterFULL.withLocale(Locale.GERMAN))); ①
System.out.println("US: " + zonedDateTime.format(dateTimeFormatterFULL.withLocale(Locale.US))); ②

System.out.println("GRM: " + zonedDateTime.format(dateTimeFormatterLONG.withLocale(Locale.GERMAN))); ③
```

```
System.out.println("US: " + zonedDateTime.format(dateTimeFormatterLONG.withLocale(Locale.US))); ④

System.out.println("GRM: " + zonedDateTime.format(dateTimeFormatterMEDIUM.withLocale(Locale.GERMAN))); ⑤
System.out.println("US: " + zonedDateTime.format(dateTimeFormatterMEDIUM.withLocale(Locale.US))); ⑥

System.out.println("GRM: " + zonedDateTime.format(dateTimeFormatterSHORT.withLocale(Locale.GERMAN))); ⑦
System.out.println("US: " + zonedDateTime.format(dateTimeFormatterSHORT.withLocale(Locale.US))); ⑧
```

① Zostanie wydrukowane: *GRM: Samstag, 15. Dezember 2001 um 16:38:52 Mitteleuropäische Normalzeit*

② Zostanie wydrukowane: *US: Saturday, December 15, 2001 at 4:38:52 PM Central European Standard Time*

③ Zostanie wydrukowane: *GRM: 15. Dezember 2001 um 16:38:52 MEZ*

④ Zostanie wydrukowane: *US: December 15, 2001 at 4:38:52 PM CET*

⑤ Zostanie wydrukowane: *GRM: 15.12.2001, 16:38:52*

⑥ Zostanie wydrukowane: *US: Dec 15, 2001, 4:38:52 PM*

⑦ Zostanie wydrukowane: *GRM: 15.12.01, 16:38*

⑧ Zostanie wydrukowane: *US: 12/15/01, 4:38 PM*

# Problemy

Jeżeli chodzi o metody typu `ofLocalizedDateTime()`, w praktyce może wydarzyć się coś takiego, że na 2 różnych komputerach będzie ustawione to samo `Locale`, a mimo to, 2 komputery drukują różne daty. W takim przypadku może to wynikać z różnic w systemach operacyjnych/wersjach Javy/vendorach Javy. Jeżeli taki problem będzie występował, można pobawić się metodą `.withLocale()`. Warto natomiast wiedzieć, że w praktyce takie rzeczy się zdarzają, pomimo, że dokumentacja określa jak powinno wyglądać poprawne zachowanie.

# Notatki - Obsługa plików a wyjątki

## Spis treści



## Try-With-Resources

Zacznijmy od fragmentu kodu, żeby mieć o czym rozmawiać. Załóżmy, że mamy 2 pliki tekstowe na dysku i chcemy napisać program, który podczas działania odczyta coś z jednego pliku i zapisze do drugiego. W przykładach pokażemy jak można to zrobić, natomiast na ten moment, nie będziemy się skupiać na wyjaśnianiu kodu, który faktycznie operuje na plikach, to będzie potem. Skupimy tylko na części dotyczącej wyjątków.

```
public void example(Path path1, Path path2) { ①
    BufferedReader in = null; ②
    BufferedWriter out = null; ③
    try {
        in = Files.newBufferedReader(path1);
        out = Files.newBufferedWriter(path2);
        String line = in.readLine();
        out.write(line);
    } catch (IOException e) { ④
        e.printStackTrace();
    } finally { ⑤
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) { ⑥
                e.printStackTrace(); ⑦
            }
        }
        if (out != null) { ⑧
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

① Path - klasa, która jest w stanie operować na ścieżce do pliku na dysku.

② BufferedReader - klasa, która pozwoli nam czytać z pliku.

③ BufferedWriter - klasa, która pozwoli nam pisać do pliku.

④ IOException - wyjątek, który może zostać wyrzucony jak coś pójdzie nie tak z odczytem lub zapisem do pliku.

⑤ Zabawa polega na tym, że musimy zadbać o to, żeby na koniec "zamknąć" otwarty plik. Z dokumentacji: *close() - closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.*

⑥ Podczas samego zamykania `.close()`, też może zostać wyrzucony `IOException`.

⑦ Dodam, że z `.close()` trzeba uważać, bo w tym przykładzie napisaliśmy je w oddzielnych `try-catch` (oddzielne dla `in` i oddzielne dla `out`), ale jeżeli byśmy zamiast robić `try-catch` dodali `throws IOException` w definicji metody, mogłaby wystąpić taka sytuacja, że pierwszy `close()` się wykonał, wyrzucił wyjątek i wtedy drugi `close()` się nie wykona, bo obsługa wyjątku ma nastąpić w metodzie wywołującej metodę `example()` i ktoś o tym zwyczajnie zapomni.

⑧ Po co w ogóle to zamykać? Żeby nie doprowadzać do wycieków pamięci, które nastąpią, bo zarezerwowaliśmy jakiś fragment pamięci, używamy go i następnie trzeba go zamknąć. Pamiętasz, że Java robi to za nas bo używa `Garbage Collectora`? Nie w tej sytuacji 😁.

Wygląda na skomplikowane, prawda? W Javie 7 zostało wprowadzone ułatwienie, które nazywa się **`try-with-resources`**. Poniższy fragment kodu robi dokładnie to samo co poprzednik. Zwróć uwagę na ciekawy zapis `try`.

```
public void tryWithResourcesExample(Path path1, Path path2) throws IOException {
    try (
        BufferedReader in = Files.newBufferedReader(path1);
        BufferedWriter out = Files.newBufferedWriter(path2)
    ) {
        out.write(in.readLine());
    }
}
```

Cały kod się uprościł, zatem przejdźmy do tego co się tutaj dzieje. W Javie 7 został wprowadzony zapis `try-with-resources`, który umożliwia nam zapisanie samego `try` (pamiętasz, że `try` musiał mieć albo `catch` albo `finally`? Nie można było napisać samego `try`? W tym przypadku można).

```
try
( ①
    BufferedReader read = Files.newBufferedReader(path1); ②
    BufferedWriter write = Files.newBufferedWriter(path2) ③
) { ④
⑤
} ⑥
```

① Nawias zwykły.

② Jeżeli definiujemy 2 zmienne, musimy oddzielić je średnikiem.

③ Tutaj średnik nie jest konieczny.

④ Koniec zwykłego nawiasu.

⑤ Kod który się wykona w obrębie `try-with-resources`.

⑥ W tym miejscu następuje automatyczne zamknięcie zasobów.

Z takim zapisem normalnie funkcjonuje zapis `catch` oraz `finally`. Nadal możemy mieć kilka `catchy` i

jedno `finally` (**nie możemy mieć kilku `finally`**)

Ważne do zapamiętania jest tutaj również to, że jeżeli zdefiniujemy jakąś zmienną w `try` w nawiasach, to możemy jej używać tylko w obrębie `try` - `catch` i `finally` już jej nie widzą. Czyli w poprzednim przykładzie zmienne `read` i `write` są widoczne tylko w zakresie bloku `try`.

# AutoCloseable

Jeżeli zaczniesz pisać przykłady na własną rękę to szybko zwrócisz uwagę, że nie można napisać czegoś takiego:

```
public void nonWorkingExample() {
    try (String variable1 = "zajavka") {

    }
}
```

Dlaczego się tak dzieje? Bo `try-with-resources` ma taki wymóg, że w obrębie nawiasu (tego zwykłego, a nie klamrowego) w `try`, możemy tworzyć tylko obiekty klas, które implementują interfejs `AutoCloseable`. `String` tego nie robi. Napisana przez Ciebie klasa też tego nie robi, dopóki nie zaimplementujesz tego jawnie (tak jak w przypadku np. `Comparable`).

Co się natomiast stanie gdy zaimplementujemy taki interfejs?

```
public class AutoCloseableExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            System.out.println("Doing something with Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        }
    }

    static class Door implements AutoCloseable {

        @Override
        // Nie musimy tu pisać throws Exception.
        // Natomiast jak jest napisane to trzeba je obsłużyć w bloku catch pod try.
        public void close() throws Exception {
            System.out.println("I'm closing my door!");
        }
    }
}
```

Musimy też wtedy zaimplementować metodę `close()` z tego interfejsu, która określa co ma się stać na końcu bloku `try-with-resources`. Na tej podstawie Java wie, co ma się stać na etapie zamykania zasobów przydzielonych na początku bloku `try-with-resources`. Inaczej mówiąc, to w metodzie `close()` piszemy w jaki sposób mają zostać zwolnione zasoby zarezerwowane w `try() {}`.

Dlaczego to działało w przypadku klas `BufferedReader` oraz `BufferedWriter` i nie dostawaliśmy błędu kompilacji? Obie te klasy implementują interfejs `AutoCloseable`.

Istnieją też 2 zalecenia co robić, a czego nie robić w metodzie `close()`:

- Nie wyrzucać wyjątku `Exception`, tylko jakiś bardziej konkretny, mówiący co faktycznie się stało.
- Tak napisać metodę `close()` aby była **`idempotentna`** (bardzo fajne słowo).

Idempotentność oznacza, że możemy tę samą metodę wywoływać ile nam się razy podoba i za każdym razem będzie to miało ten sam efekt. Czy wywołamy ją po raz pierwszy czy 14, zawsze efekt powinien być ten sam. Inaczej mówiąc, taka metoda nie ma efektów ubocznych. Taka rekomendacja pojawia się żebyśmy nie zrobili sobie kuku jeżeli `close()` zostanie wywołane 2 razy.

W ramach ciekawostki dodam, że istnieje taki interface jak `Closeable`, który istniał przed `AutoCloseable`. Jak zaczniesz oglądać jak wzajemnie dziedziczą z siebie te interfejsy, to zobaczysz, że `Closeable` dziedziczy z `AutoCloseable`. To był taki trick, żeby zachować kompatybilność wsteczną, która jest przecież jedną z myśli przewodnich Javy. Możesz zwrócić uwagę, że w definicji `Closeable` w metodzie `close()` jest napisane `throws IOException`, podczas gdy `AutoCloseable` ma już bardziej poluźnione reguły i pozwala na wyrzucenie samego `Exception`. Przypomnę, że `IOException`, które jest rzucane przy sytuacjach wyjątkowych, podczas operacji na plikach, dziedziczy z `Exception`.

# Suppressed exceptions

Może nastąpić taka sytuacja, że w trakcie wywołania metody `close()` zostanie wyrzucony wyjątek. Dla jasności jeszcze raz ten sam fragment kodu.

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            System.out.println("Doing something with Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
            throw new RuntimeException("Can't close my Door!");
        }
    }
}
```

A co jeżeli w środku bloku `try`, też zostanie wyrzucony wyjątek?

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            throw new RuntimeException("Exception while opening Door");
        } catch (Exception e) {
            System.out.println("Handling exception thrown by close(): " +  e.getMessage());
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
          throw new RuntimeException("Can't close my Door!");
        }
    }
}
```

Jeżeli wystąpi sytuacja jak powyżej, wywołanie kodu:

```
throw new RuntimeException("Exception while opening Door");
```

powoduje zatrzymanie wywołania bloku `try` i przejście do wywołania metody `.close()`. Ale przecież metoda `close()` również wyrzuca wyjątek. Nazywany jest on `SuppressedException`. Na ekranie wydrukowany wtedy zostaje wyjątek główny, czyli wywołanie kodu:

```
System.out.println("Handling exception thrown by close(): " +  e.getMessage());
```

Drukuje na ekranie:

```
Handling exception thrown by close(): Exception while opening Door
```

Ale, w ten sposób, nie widzimy jaki wyjątek został wyrzucony w metodzie `close()`. Jeżeli natomiast napiszemy w catch `e.printStackTrace()`, dostaniemy taki (albo zbliżony) StackTrace na ekranie:

```
java.lang.RuntimeException: Exception while opening Door
    at SuppressedExample.main(SuppressedExample.java:13)
Suppressed: java.lang.RuntimeException: Can't close my Door!
    at SuppressedExample$Door.close(SuppressedExample.java:26)
    at SuppressedExample.main(SuppressedExample.java:12)
```

Zauważ, że `StackTrace`, również nazywa wyjątek wyrzucony w metodzie `close()` jako `Suppressed`. Aby się do niego dostać, można napisać kod w ten sposób:

```
public class SuppressedExample {

    public static void main(String[] args) {
        try (Door door = new Door()) {
            throw new RuntimeException("Exception while opening Door");
        } catch (Exception e) {
            for (Throwable throwable: e.getSuppressed()) {
                System.out.println(throwable.getMessage());
            }
        } finally {
            System.out.println("Calling finally");
        }
    }

    static class Door implements AutoCloseable {

        @Override
        public void close() throws Exception {
          throw new RuntimeException("Can't close my Door!");
        }
    }
}
```

# Notatki - System plików

# Spis treści



# File system

Zanim wejdziemy w temat plików, powiedzmy sobie czym jest plik, a czym jest katalog.

## Plik

Plik (*file*) jest zbiorem danych, który przechowuje informacje użytkownika albo systemu operacyjnego. Plik zawiera w sobie skończoną ilość danych oraz posiada atrybuty, takie jak np. rozmiar, data utworzenia, data modyfikacji. Pliki są umiejscowione w katalogach (folderach).

## Katalog

Katalog (*directory*) jest bytem w systemie operacyjnym, który może zawierać pliki albo inne foldery. W kodzie, do folderów często odwołujemy się identycznie jak do plików, wynika to z tego, że mają one podobne atrybuty, jak chociażby data utworzenia. Należy również pamiętać, że zarówno pliki jak i foldery znajdujące się w tym samym folderze muszą mieć unikalną nazwę.

W nomenklaturze istnieje takie pojęcie jak **root**. Jest to najwyższy folder, do którego możemy się odwołać. W przypadku Windowsa, może to być `C:\`.

## File system

System plików (*file system*) - jest on odpowiedzialny za odczyt i zapis plików na komputerze. Różne systemy operacyjne korzystają z różnych systemów plików. Klasycznym przykładem jest `Windows` i `Linux`, które korzystają z różnych systemów plików. Całe szczęście Java dostarcza nam API, dzięki któremu możemy operować na plikach w różnych systemach plików. Pozwala nam to na wykorzystanie tego

samego kodu do operowania na różnych systemach operacyjnych.

Nie zagłębiamy się w różnice pomiędzy systemami plików używanych w konkretnych systemach operacyjnych. Ważne jest natomiast dla nas to, że jak napiszemy kod w Javie, to może on być uruchomiony zarówno na `Windows` jak i na `Linux`, bo Java poradzi sobie z tymi systemami plików.

# Ścieżka

Ścieżka (*path*) jest reprezentacją pliku albo katalogu w systemie plików w taki sposób, że wiemy konkretnie do jakiego pliku lub katalogu się odnosimy. Coś jak adres mieszkania. W Windowsie ścieżki są zapisywane z backslashem `\`, a na Linux ze zwykłym slashem `/`.

Mieliśmy już z tym wszystkim do czynienia wcześniej, podczas gdy tworzyliśmy klasy i paczki w Javie. Tworząc klasę, Intellij tworzy nam na dysku plik `NazwaKlasy.java`, tworząc paczkę natomiast, tworzymy na dysku katalog z nazwą paczki.

# Absolute path

Ścieżka absolutna to pełna ścieżka do pliku zaczynająca się od roota, uwzględniając wszystkie katalogi, które znajdują się po drodze od roota do naszego pliku lub katalogu.

*Przykład ścieżki absolutnej*

```
C:\Users\zajavka\work\bootcamp\Main.java
```

# Relative path

Ścieżka relatywna to ścieżka do naszego pliku, relatywnie odnosząca się do innego katalogu.

*Przykład ścieżki relatywnej*

```
bootcamp\Main.java
```

Ścieżka taka może tak na prawdę odnosić się do kilku różnych miejsc, bo `bootcamp\Main.java` może istnieć w wielu katalogach na dysku. W przypadku ścieżki absolutnej, mówimy np. o jednym konkretnym unikalnym pliku.

# Separator

Przypomnijmy sobie co zostało wspomniane wcześniej. `Windows` i `Linux` mają inne separatory katalogów/plików. W Windows ścieżki są zapisywane z backslashem `\`, a na Linux ze zwykłym slashem `/`.

Java może nam podać informację, co jest separatorem w sposób pokazany poniżej:

```
// Można tak
System.out.println(System.getProperty("file.separator"));
// Albo tak
System.out.println(java.io.File.separator);
```

W programie Javowym do zapisu ścieżki, możemy używać separatora `/`. Java sobie dalej z tym poradzi, nawet gdy używamy Windowsa, w którym separatorem jest `\`.

## Symbole charakterystyczne

Ogólnie jest przyjęte stosowanie 2 charakterystycznych symboli na ścieżkach do plików.

- **.** - (pojedyncza kropka) - oznacza odniesienie do katalogu, w którym znajdujemy się obecnie
- **..** - (podwójna kropka) - oznacza odniesienie do katalogu rodzica, czyli katalogu, w którym znajduje się nasz katalog

Czyli poniższy przykład:

```
../someDirectory/someFile.txt
```

Oznacza, że z katalogu, w którym znajdujemy się obecnie, wyjdziemy do katalogu rodzica (do katalogu wyżej) i tam poszukamy folderu `someDirectory` z plikiem `someFile.txt`.

Możemy też zapisywać bardziej złożone kombinacje, np:

```
../../../someFile.txt
```

Co oznacza, że 3 razy staramy się wyjść "do góry" w stosunku do katalogu, w którym znajdujemy się obecnie i tam poszukamy pliku `someFile.txt`

Natomiast taki zapis:

```
./someFile.txt
```

Oznacza, że odnosimy się do pliku `someFile.txt` w katalogu, w którym znajdujemy się obecnie.

# Klasa File

Bardzo często będziemy używać klasy `java.io.File`, dlatego to od niej zaczniemy. Klasa ta jest używana do odczytywania informacji o plikach, tworzenia lub usuwania plików. Jednocześnie klasa ta nie może modyfikować zawartości pliku - do tego są inne sposoby.

Jest to dosyć nieintuicyjne dlatego chcemy to zaznaczyć. Klasa `File` może operować zarówno na plikach jak i katalogach.

# Użycie obiektu klasy File

```
public class FileExamples {

    public static void main(String[] args) {
        File file1 = new File("myFileInDir.txt");
        System.out.println(file1.exists());
        File file2 = new File("myFolder/myFileInDir.txt");
        System.out.println(file2.exists());
        File file3 = new File("myAnotherFolder/myFolder/myFileInDir.txt");
        System.out.println(file3.exists());
    }
}
```

Gdy uruchamiamy nasz program z poziomu Intellij, możemy zdefiniować ścieżkę relatywną i plik zostaje znaleziony - pod warunkiem, że ścieżka relatywna zaczyna się od roota naszego projektu. Dlatego z jednej strony bezpieczniej jest posługiwać się ścieżką absolutną. Ale gdy taki kod jest uruchamiany na różnych komputerach, to przecież ta ścieżka absolutna może być inna. Dlatego najczęściej używa się ścieżek relatywnych w odniesieniu do lokalizacji uruchamianego programu.

Pamiętaj - obiekt `File` w Javie możemy utworzyć gdy plik fizycznie nie istnieje na dysku. Możemy taki obiekt również utworzyć, gdy taki plik na dysku istnieje.

# Popularne metody z klasy File

| Nazwa metody | Opis |
|---|---|
| exists() | Sprawdza, czy plik istnieje |
| isDirectory() | Sprawdza czy obiekt File jest katalogiem |
| isFile() | Sprawdza czy obiekt File jest plikiem |
| length() | Zwraca ilość bajtów w pliku. Może wystąpić taka sytuacja, że plik zarezerwuje na dysku więcej bajtów niż faktycznie jest potrzebne, kwestia performance |
| lastModified() | Zwraca ilość milisekund od Epoch do momentu, gdy plik był edytowany po raz ostatni |
| delete() | Usuwa z dysku plik albo katalog. Aby usunąć katalog, musi on być pusty |
| renameTo(File) | Zmienia nazwę pliku na podaną jako argument |
| mkdir() | Tworzy katalog, pod ścieżką określoną w obiekcie File |
| mkdirs() | Jeżeli mamy na podanej ścieżce kilka nieistniejących katalogów, zostaną stworzone wszystkie nieistniejące |
| getName() | Zwraca nazwę pliku |
| getAbsolutePath() | Zwraca ścieżkę absolutną |
| getParent() | Zwraca ścieżkę rodzica albo null jeżeli takiego rodzica nie ma |
| listFiles() | Zwraca tablicę File[] z plikami lub katalogami w danym katalogu |

Poniżej przykład wykorzystania niektórych metod z tabeli:

```
public class FileExamples {

    void example() {
        File file = new File("someFile.txt");
        if (!file.exists()) {
            System.out.println("File: " + file.toString() + " doesn't exist");
            return;
        }

        System.out.println("File: " + file.toString() + " exists");
        System.out.println("file.getAbsolutePath(): " + file.getAbsolutePath());
        System.out.println("file.getParent(): " + file.getParent());

        if (file.isFile()) {
            System.out.println("File: " + file.toString() + " is file");
            System.out.println("file.length(): " + file.length());
            return;
        }

        if (file.isDirectory()) {
            System.out.println("File: " + file.toString() + " is directory");
            for (File subfile : file.listFiles()) {
                System.out.println("subfile.getName(): " + subfile.getName());
            }
        }
    }
}
```

# Notatki - Streamy - BufferedInputStream i BufferedOutputStream

## Spis treści



## BufferedInputStream i BufferedOutputStream

Czyli jak poprawić wydajność poprzedniego rozwiązania. Wystarczy powyższe `FileInputStream` oraz `FileOutputStream` opakować w `BufferedStream`. W takim podejściu zamiast operować na pojedynczych bajtach, zaczniemy operować na tablicy bajtów.

*Klasa StreamsExamples*

```
public class StreamsExamples {

    public static void main(String[] args) throws IOException {
        File inputFile = new File("myInputFile.txt");
        File outputFile = new File("myOutputFile.txt");
        justCopyWithBuffer(inputFile, outputFile);
    }

    private static void justCopyWithBuffer(File source, File destination) throws IOException {
        try (
            InputStream input = new BufferedInputStream(new FileInputStream(source));
            OutputStream output = new BufferedOutputStream(new FileOutputStream(destination))
        ) {
            byte[] buffer = new byte[1024];
            int lengthRead = input.read(buffer);
            System.out.printf("Starting buffered reading file: [%s]%n", source);
            System.out.printf(
                "Read value: [%s], chars: [%s], length: [%s]%n",
                PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer), lengthRead);

            while (lengthRead > 0) {
                System.out.printf(
                    "Writing buffered file value: [%s], char: [%s]%n",
                    PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer));

                output.write(buffer, 0, lengthRead);
                output.flush();
                lengthRead = input.read(buffer);

                System.out.printf(
                    "Read buffered file value: [%s], char: [%s], length: [%s]%n",
                    PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer), lengthRead);
            }
        }
    }
}
```

*Klasa PrintingUtils*

```
public class PrintingUtils {

    static String toCharString(byte[] input) {
        char[] charArray = new char[input.length];
        for (int i = 0; i < input.length; i++) {
            charArray[i] = (char) input[i];
        }
        return replaceNewLines(Arrays.toString(charArray));
    }

    static String byteArrToStr(byte[] buffer) {
        return replaceNewLines(Arrays.toString(buffer));
    }

    static String replaceNewLines(String input) {
        return input.replace("\n", "\\n").replace("\r", "\\r");
    }
}
```

Zamiast odczytywać bajt bo bajcie, używamy metody `read(byte[])`, która jako wynik zwraca nam ilość odczytanych bajtów z pliku i zapisanych w tablicy `byte[]`. Ta zwracana wartość jest o tyle istotna, że jeżeli metoda `read(byte[])` zwróci wartość `<= 0`, oznacza to koniec pliku. Pamiętać również należy o tym, że gdy dojdziemy do końca odczytywanego pliku, tablica `byte[]` tylko częściowo wypełni się danymi, które nas interesują. Czyli jeżeli tablica `byte[]` ma rozmiar `1024`, a w pliku zostało tylko `24` bajty do odczytu, to kolejne `1000` bajtów zostanie wypełnione wartościami z poprzednich odczytów. Po to właśnie mamy wartość `length` zwracaną przy odczycie, żebyśmy wiedzieli ile faktycznie odczytaliśmy podczas tego przebiegu przydatnych bajtów. Resztę możemy wtedy spokojnie pominąć.

Metoda `write()` jest o tyle ciekawa, że przyjmuje 3 argumenty:

- **faktyczną tablicę z bajtami do zapisu**,
- **offset** - ilość bajtów, którą możemy pominąć przy zapisie, najczęściej jednak wstawia się tu 0,
- **length** - ilość bajtów z tablicy, którą chcemy zapisać (znowu przydaje się tutaj w naszym przykładzie długość tablicy odczytanej, bo przy ostatnim zapisie `output.write(buffer, 0, lengthRead);` możemy określić ile bajtów z tej tablicy faktycznie zapiszemy, odrzucając te, które jak wspomniałem wcześniej, są uzupełnione wartościami z poprzednich odczytów)

I jeszcze raz, używanie klas z `Buffered` w nazwie, poprawia wydajność aplikacji ☺. W tym przykładzie dodano też metodę `flush()`, która wymusza faktyczny zapis do pliku wypełnimy w kodzie naszą tablicę z bajtami ponownie. O metodzie `flush()` wspominaliśmy na etapie omawiania teorii Streamów.

Jeszcze jedna dygresja. Mówiliśmy o poprawie wydajności przy przejściu z `FileInputStream` na `BufferedInputStream`. Można też próbować poprawić wydajność określając rozmiar tablicy `byte[]`. Najczęściej stosuje się rozmiar `1024`. Jednakże największą różnicę zobaczymy przesiadając się z `FileInputStream` na `BufferedInputStream`. Chociaż tutaj można też bawić się różnymi rozmiarami tablicy żeby zobaczyć jak wpłynie to na czas zapisu i odczytu.

W Javie 9 została wprowadzona metoda `readAllBytes()`, która pozwala na odczyt wszystkich pozostałych w Streamie bajtów. Należy jednak pamiętać, że dokumentacja tej metody wspomina, że jest ona dedykowana do małych plików, gdzie można wczytać całą zawartość pliku do tablicy z bajtami "na raz".

Nie jest ona przeznaczona do odczytu dużych plików. Do prawidłowego stosowania tej metody polecam zapoznać się z dokumentacją.

## Po co w sumie są BufferedStreamy?

Systemy plików są tworzone w taki sposób, że są wydajne, gdy staramy się dostać do plików na dysku w sposób sekwencyjny. Im więcej bajtów staramy się odczytać w sekwencji jednocześnie, tym mniej operacji odczytu musimy wykonać. W ten sposób optymalizujemy naszą aplikację, bo mniej musi ona wykonać faktycznych odwołań do dysku w celu wykonania jakiejś operacji na pliku (np. odczyt). Przykładowo odczyt 8 bajtów w sekwencji będzie szybszy niż odczyt 8 bajtów rozmieszczonych losowo po dysku komputera.

# Notatki - Streamy - FileInputStream oraz FileOutputStream

## Spis treści



## FileInputStream oraz FileOutputStream

Wymienione klasy (`FileInputStream` oraz `FileOutputStream`) są najbardziej podstawowymi, o których będziemy rozmawiać. Służą do odczytu bajtów z pliku oraz do zapisu bajtów do pliku. Odczyt z pliku następuje aż do odczytania wartości `-1`. Gdy napotkamy taką wartość, należy przerwać odczyt z pliku, oznacza to jego koniec.

Przykład w kodzie:

*Klasa StreamsExamples*

```
public class StreamsExamples {

    public static void main(String[] args) throws IOException {
        File inputFile = new File("src/pl/zajavka/java/myInputFile.txt");
        File outputFile = new File("src/pl/zajavka/java/myOutputFile.txt");
        CopyExample.justCopyNoBuffer(inputFile, outputFile);
    }

}
```

*Klasa CopyExample*

```
public class CopyExample {

    static void justCopyNoBuffer(File source, File destination) throws IOException {
        try (
            InputStream input = new FileInputStream(source);
            OutputStream output = new FileOutputStream(destination)
        ) {
            int value = input.read();
            System.out.printf("Starting reading file: [%s]%n", source);
            System.out.printf("Read value: [%s], char: [%s]%n", value, (char) value);

            while (value != -1) {
                System.out.printf("Writing value: [%s], char: [%s]%n", value, (char) value);
                output.write(value);
                value = input.read();
                System.out.printf("Read value: [%s], char: [%s]%n", value, (char) value);
            }
        }
    }
}
```

Jeżeli chcemy zobaczyć jakie znaki stoją pod wczytanymi bajtami, możemy odwołać się do tablicy `Unicode` (zaraz zostanie opisane, że nie zawsze to zadziała). Możemy wtedy próbować rzutować otrzymaną wartość liczbową na `char` żeby zobaczyć jaki to znak (sposób ten działa dla np. angielskich liter, więcej poniżej).

Pamiętać również należy, że przykład powyżej jest wolny pod względem wydajności bo operujemy na pojedynczych wartościach bajtów.

Sytuacja opisana powyżej będzie działała poprawnie np. dla **nie**polskich znaków. Jeżeli chcemy wygodnie operować na polskich znakach musimy przejść do klas "...Reader" oraz "...Writer" i zastosować wspomniane wcześniej kodowanie znaków.

Po przeanalizowaniu kodu powyżej zwróć uwagę, że odczytywane wartości są typu `int` a nie `byte`. Wynika to z tego, że zakres `byte` to (-128;127), podczas gdy my potrzebujemy odczytywać wartości w zakresie `(0;255)`. Druga sprawa jest taka, że używając typu `byte` pozbawiamy się możliwości oznaczenia `-1` reprezentującej koniec pliku (`EOF` - end of file), gdyż `-1` jest poprawną wartością typu `byte`. Dlatego też, odczytywane wartości są typu `int`.

Należy do tego wszystkiego dodać jeszcze dwa terminy:

- signed variables (zmienne ze znakiem) - oznacza wartości, które mogą być reprezentowane zarówno przez liczby dodatnie jak i ujemne. Możemy zatem określić stwierdzenia takie jak: `signed byte` lub `signed int`.
- unsigned variables (zmienne bez znaku) - oznacza wartości, które mogę być reprezentowane tylko przez liczby dodatnie oraz zero. Możemy zatem określić stwierdzenia takie jak: `unsigned byte` lub `unsigned int`.

Dlaczego ma to tutaj znaczenie? Wcześniej zostało napisane, że w kodzie powyżej wczytywany jest `int` a nie `byte` oraz wyjaśnione dlaczego. Jeżeli natomiast spróbujemy wykonać kod poniżej:

```
byte[] bytes = "ą".getBytes(StandardCharsets.UTF_8);
System.out.println(Arrays.toString(bytes));
```

To na ekranie zostanie wydrukowane `[-60, -123]`.

Dlaczego tak kręcę z tymi przykładami? Kodowanie znaków `UTF-8` może wykorzystać od 1 do 4 bajtów żeby zakodować jakiś znak. Na przykładzie literki `ą` widać, że wykorzystane zostały 2 bajty w celu jej zakodowania. Kodowanie znaków `UTF-8` pozwala nam określić, że dane dwa bajty mają reprezentować literkę `ą`. Jeżeli tę samą literkę `ą` zapiszemy w pliku tekstowym i spróbujemy ją wczytać stosując kod metody `justCopyNoBuffer()` to okaże się, że dostajemy wartości `196` oraz `133`. Metoda `justCopyNoBuffer()` z powodzeniem zapisze do innego pliku tę samą treść i będzie się ona poprawnie wyświetlała po zapisie w pliku, natomiast jak spróbujemy rzutować podane dwie wartości po ich wczytaniu na `char` to zgodnie z tablicą `Unicode` dostaniemy jakieś krzaki. Z tego można wyciągnąć dwa wnioski:

- `FileInputStream` oraz `FileOutputStream` po prostu czyta bajty, nie zastanawia się co dalej z tymi bajtami zrobić. Dopiero kodowanie `UTF-8` pozwala nam zinterpretować, że wczytane dwa bajty mają reprezentować literkę `ą`. Dlatego o tym wspominam, żeby później móc się do tego odwołać w przykładach z klasami "...Reader" oraz "...Writer".
- Przy odczycie danych za pomocą metody `read()` wartości wczytanych z pliku bajtów zostały zamienione na zakres (0;255). Zatem przy rzutowaniu tych wartości na `char` dostajemy dwa znaki stojące pod wartościami `196` oraz `133` zamiast jednego `ą`. Jeżeli chcielibyśmy, żeby te dwie wartości miały sensowne dla nas znaczenie należy zastosować kodowanie `UTF-8` i w tym celu trzeba użyć klas "...Reader" oraz "...Writer".

Kodowanie `UTF-8` było podawane jako przykład, przypominam, że nie jest to jedyny możliwy sposób kodowania.

Nie skupiamy się na algorytmie zamiany `signed byte` na `unsigned byte`. Jeżeli ktoś jest tym zainteresowany, zachęcam do pogrzebania w internecie 😀.

# Notatki - Streamy - ObjectInputStream i ObjectOutputStream

## Spis treści



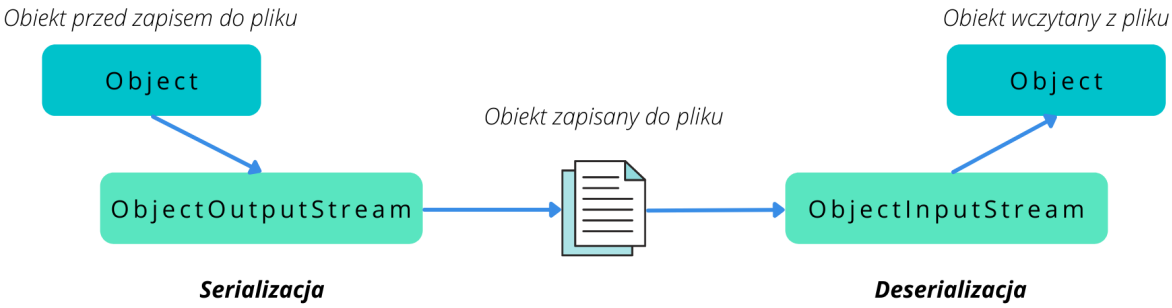
## ObjectInputStream i ObjectOutputStream

A może chcemy w trakcie działania programu zapisać nasz cały obiekt do pliku a potem go wczytać?

Proces konwersji obiektu, który mamy w pamięci do jego reprezentacji pozwalającej go zapisać (np. do pliku) nazywamy `serializacją` obiektu. Proces odwrotny, czyli wczytanie obiektu (np. z pliku) do pamięci programu nazywamy `deserializacją` obiektu.

Sama `serializacja` i `deserializacja` w praktyce nie jest może często używana w tej formie, gdzie chcemy zapisać obiekt do pliku aby móc go wczytać gdzieś w przyszłości. Możesz natomiast spotkać się w praktyce z taką sytuacją gdzie pobierzesz dane z jakiegoś systemu zewnętrznego, zapiszesz je do jakiegoś obiektu w pamięci i będziesz potrzebować zapisać gdzieś ten obiekt razem z jego wszystkimi danymi. Przykładowo można wtedy zapisać taki obiekt w bazie danych, z której możemy szybciej pobrać dane niż z systemu zewnętrznego. Dzięki temu takie dane będą o wiele szybciej dostępne w Twoim programie i nie będzie trzeba ich pobierać ponownie z jakiegoś systemu zewnętrznego. Oczywiście przedstawiona sytuacja jest przykładowa 😀.

Swoją drogą to sytuacja opisana powyżej nazywana jest `Cacheowaniem danych`, a samo miejsce gdzie takie dane są przetrzymywane nazywane jest `Cache`. Czyli od teraz jak powiemy, że coś zostało "Skeszowane", to już wiesz, że dane zostały gdzieś zapisane w takiej formie, żebyśmy mieli do nich bardzo szybki dostęp niż gdybyśmy pobierali te dane bezpośrednio ze źródła.




*Obraz 1. Schemat procesu serializacji i deserializacji*

# Serializable

`java.io.Serializable` jest interfejsem potrzebnym do tego, żeby móc przeprowadzić serializację obiektu. Jest to tak zwany **`marker interface`**, czyli interface oznaczający, że z danym obiektem ma się coś stać. Taki interfejs nie ma metod. Oznacza to, że jeżeli chcemy serializować nasz obiekt, klasa na podstawie której został stworzony, musi implementować interfejs `Serializable`. Klasy wbudowane w API Javy często implementują ten interface domyślnie. Jeżeli nasz obiekt ma pola, które są zdefiniowane przez klasy napisane przez nas, również te klasy muszą implementować interfejs `Serializable`.

Czyli jeżeli mamy obiekt samochód, który ma koła oraz drzwi i chcemy ten samochód zserializować, to zarówno samochód, koło oraz drzwi muszą implementować interfejs `Serializable`. Inaczej dostaniemy wyjątek `NotSerializableException` w trakcie działania programu.

## transient

Jeżeli natomiast chcielibyśmy aby w naszym samochodzie serializowalne były koła, ale drzwi już nie, musielibyśmy oznaczyć pole 'drzwi' jako `transient`. Czyli tak samo jak w definicji pola możemy wpisać `final`, tak samo możemy wpisać `transient`. W takim przypadku Java będzie wiedziała, że drzwi są do pominięcia w procesie serializacji.

Do tego należy dodać, że pola statyczne również nie będą serializowane, co jest całkiem logiczne, bo przecież statyczny oznacza, że nie przynależy do obiektu, tylko do klasy. Serializujemy obiekty, a nie klasy.

## serialVersionUID

Do całego procesu serializacji/deserializacji należy jeszcze dołożyć taką zmienną jak:

```
private static final long serialVersionUID = 1L;
```

Jeżeli chcemy oznaczyć, że nasz obiekt może być serializowalny, konieczne jest tylko implementowanie przez ten obiekt interfejsu `Serializable`. Pole `serialVersionUID` nie jest konieczne, jest to natomiast dobra praktyka, żeby zdefiniować to pole i aktualizować jego wartość za każdym razem gdy zmieniamy schemat naszej klasy (dodajemy, usuwamy lub modyfikujemy pola klasy).

Pole `serialVersionUID` oznacza wersję naszej klasy, w której obiekt był serializowany. Tak jak mamy wydania książki 1, 3, 5 i 9. Podczas serializowania obiektu, zapisywana jest również informacja o

`serialVersionUID`, dzięki czemu podczas procesu deserializacji jesteśmy w stanie stwierdzić, czy podczas serializowania mieliśmy tę samą wersję klasy jak podczas deserializacji. Jeżeli wersje się nie zgadzają, możemy szybko wykryć, że będziemy mieli problem z deserializacją.

Jeżeli sami nie zadeklarujemy tego pola podając jawnie, zostanie ono wygenerowane automatycznie, a wartość zostanie określona na podstawie "różnych aspektów klasy" (tak fajnie opisuje to dokumentacja Oracle Serializable Docs).

Jednakże mocno zaleca się generowanie tej wartości samodzielnie, gdyż jeżeli będziemy generować ją automatycznie może to prowadzić do dziwnych, trudnych do ogarnięcia błędów. Takie rozjazdy mogą się pojawić np. przy innych kompilatorach. Zaleca się też aby to pole było definiowane jako `private`.

Co do wartości pola `serialVersionUID`, często zdarza się że jego wartość jest ustawiona na `1`. Jest to robione po to, żeby przy dodawaniu kolejnych pól do klasy można było próbować deserializować obiekty, które zostały zserializowane przed dodaniem tych pól. Gdybyśmy używali wartości pola `serialVersionUID`, które byłby unikalne i zmieniane przy każdej edycji klasy, nie bylibyśmy w stanie deserializować 'starego' obiektu po dodaniu do klasy nowych pól.

## Jakie klasy serializować?

W nagłówku zostało zadane dobre pytanie. Dodajmy do tego jeszcze czemu w zasadzie wszystkie klasy nie są automatycznie oznaczane jako `Serializable`? Powodem jest to, że mogą wystąpić klasy (a nawet występują niektóre klasy wbudowane), których nie chcemy serializować. Są to takie klasy, które byłoby bardzo ciężko serializować, a czasem mogłoby być to wręcz niemożliwe. Przykładem mogą być klasy Streamów. Stosując to podejście, deweloperzy muszą zatem jawnie określić które klasy chcą serializować. Czyli muszą zrobić to świadomie z pełnym rozumieniem konsekwencji tego co robią 😀.

Zagłębiając się w tematykę programowania zobaczysz, że generalnie w kodowaniu jest dużo takich zagadnień filozoficznych, czy robić coś domyślnie, czy wymusić na programistach określenie czegoś w sposób jawny.

## Nareszcie przykład

*Klasa Car*

```
public class Car implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String name;

    private final Long year;

    // jeżeli pole byśmy oznaczyli jako transient, nie zostałoby ono zserializowane
    private final List<Door> doors;

    // ... konstruktor, gettery i toString
}
```

*Klasa Door*

```
public class Door implements Serializable {

    private static final long serialVersionUID = 1L;

    private final String which;

    // ... konstruktor, gettery i toString
}
```

*Klasa ObjectSerializer*

```
public class ObjectSerializer {

    public static void serializeCars(List<Car> cars, File dataFile) throws IOException {
        try (ObjectOutputStream outputStream = new ObjectOutputStream(
            new BufferedOutputStream(
                new FileOutputStream(dataFile)))
        ) {
            for (Car car : cars) {
                outputStream.writeObject(car);
            }
        }
    }

    public static List<Car> deserializeCars(File dataFile) throws IOException, ClassNotFoundException {
        List<Car> cars = new ArrayList<Car>();
        try (ObjectInputStream inputStream = new ObjectInputStream(
            new BufferedInputStream(
                new FileInputStream(dataFile)))
        ) {
            while (true) {
                Object objectRead = inputStream.readObject();
                if (!(objectRead instanceof Car)) {
                    System.out.println("Object is not Car");
                }
                cars.add((Car) objectRead);
            }
            // jak dojdziemy do końca pliku, zostanie wyrzucony EOFException
        } catch (EOFException e) {
            System.out.println("end of file");
        }
        return cars;
    }

}
```

*Klasa ObjectStreamExample*

```
public class ObjectStreamExample {

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        List<Car> cars = new ArrayList<Car>();
        cars.add(new Car("Ford Mustang", 1969L, List.of(new Door("left"), new Door("right"))));
        cars.add(new Car("BMW 5", 2015L, List.of(new Door("left"), new Door("right"))));
        cars.add(new Car("Mercedes G-class", 2004L, List.of(new Door("left"), new Door("right"))));

        // rozszerzenie pliku nie ma znaczenia
        File destination = new File("car.whatever");

        ObjectSerializer.serializeCars(cars, destination);
        List<Car> carsDeserialized = ObjectSerializer.deserializeCars(destination);
        System.out.println(carsDeserialized);
    }
}
```

Należy w tym miejscu też dodać, że referencje `null` również mogą podlegać serializacji i deserializacji.

# Tworzenie obiektu przy deserializacji

Jak dokonujemy deserializacji to te obiekty przecież muszą się jakoś stworzyć. I tutaj dzieje się coś niespodziewanego, gdyż podczas tworzenia obiektu w procesie deserializacji nie jest wywoływany żaden konstruktor z naszej klasy serializowanej. Pola, które nie były serializowane są inicjowane wartościami domyślnymi, `null`, `0` itp. Natomiast nie jest wywoływany żaden konstruktor z naszej serializowanej klasy. Żeby być precyzyjnym, dokładnie to jest wołany bezargumentowy konstruktor pierwszej superklasy w hierarchii, która nie jest serializowalna, czyli w tym przypadku - `Object`.

W przypadku pól deserializowanych, początkowo są one inicjowane wartościami domyślnymi, a potem te wartości są nadpisywane wartościami deserializowanymi.

W filmach był pokazany przykład z `Kotem` i `Persem`. `Pers` jako klasa serializowalna nie miał wywoływanych konstruktorów na etapie deserializacji, natomiast klasa `Cat`, jako pierwsza w hierarchii, która nie jest `Serializable`, miała normalnie wywołany blok inicjalizacyjny i konstruktory.

Java natomiast odtwarza wartości pól dla obiektu serializowanego na podstawie danych, które zostały zapisane podczas serializacji.

Zmienne statyczne nie są serializowalne, zatem ich wartości są przypisywane podczas inicjowania klasy.

# Notatki - Streamy - FileReader oraz FileWriter

## Spis treści



## FileReader oraz FileWriter

`Reader` oraz `Writer` różnią się tym od `InputStream` oraz `OutputStream`, że pozwalają nam operować od razu na ciągach znaków. Kolejna rzecz to możliwość określenia kodowania jakie ma być użyte przy odczycie znaków z pliku. `Reader` oraz `Writer` są dedykowane do pracy na plikach tekstowych. Ich użycie jest bardzo podobne do `InputStream` oraz `OutputStream`.

Pokazujemy od razu przykład z `Buffered` bo w praktyce zależy nam przecież na wydajności 😀.

```
public class ReaderWriterExamples {

    public static void main(String[] args) throws IOException {
        File inputFile = new File("myInputFile.txt");
        File outputFile = new File("myOutputFile.txt");

        List<String> fileRead = readFile(inputFile);
        for (String line : fileRead) {
            System.out.println(line);
        }
        writeFile(fileRead, outputFile);
    }

    public static List<String> readFile(File inputFile) throws IOException {
        List<String> result = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
            String line = reader.readLine();
            while (line != null) {
                result.add(line);
                line = reader.readLine();
            }
        }
        return result;
    }

    public static void writeFile(List<String> fileRead, File outputFile) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
            for (String line : fileRead) {
                writer.write(line);
                writer.newLine();
            }
        }
    }
}
```

Zwróćmy uwagę na parę kwestii:

- metoda `readLine()` zwraca `String`, a nie `int`. W związku z tym, `readLine()` w przypadku końca pliku nie zwróci `-1`, tylko `null`.
- przedstawiony kod jest wygodniejszy jeżeli chcemy operować na odczytanych wartościach. Nie trzeba ich rzutować na `char` i wczytujemy całe linijki na raz. Nie mówiąc już o kodowaniu polskich znaków i możliwości łatwej pracy na nich.
- w tym przypadku możemy zapisywać całą linijkę tekstu jednocześnie.

Widać już, że zarówno poprzednie przykłady jak i ten mogą służyć do tego samego, natomiast w przypadku operowania na danych tekstowych - `Reader` i `Writer` są wygodniejsze w użyciu.

# Character encoding

Wracamy znowu do tematu kodowania znaków. Przypomnijmy sobie, że kodowanie znaków określa w jaki sposób znaki są kodowane i przechowywane w reprezentacji bajtowej i w jaki sposób z bajtów są one dekodowane z powrotem jako znaki.

Dlaczego ten aspekt jest tutaj istotny? Przypomnij sobie przykład z polskimi znakami i odczytem pliku

bezpośrednio przez `FileInputStream`. Dlatego właśnie `Reader` oraz `Writer` są wygodniejsze przy operacjach na plikach tekstowych.

# Notatki - Streamy - Teoria

# Spis treści



# Streamy

## Ale Streamy już przecież były...

To nie są te same streamy co w programowaniu funkcyjnym, pomimo tego, że nazwy są mylące ☺. Pod tym linkiem znajdziesz przykładową odpowiedź ze Stackoverflow wyjaśniającą tę kwestię. Możemy w niej przeczytać, że pechowo wyszło, że zarówno Streamy IO oraz Streamy, które poznaliśmy w programowaniu funkcyjnym nazywają się tak samo. IO Streamy (czyli te, o których zaraz będziemy rozmawiać) służą do operowania na zewnętrznych zasobach (np. plikach na dysku). Streamy w programowaniu funkcyjnym służą do przetwarzania danych w sekwencji. Koncepcje te należy rozumieć niezależnie od siebie.

Możliwe jest jednak używanie tych mechanizmów razem, np. klasa `BufferedReader` posiada zdefiniowane metody, które mogą zwracać `Stream<String>`, ale do tego przejdziemy później.

Podobieństwo jakie możemy wyróżnić pomiędzy IO Stream oraz "functional" Stream to abstrakcja, dzięki której możemy mówić o sekwencji danych. Zarówno jedna i druga koncepcja Streamów odnosi się do sekwencji danych, która to sekwencja ma jakiś początek i jakiś koniec przetwarzania.

Przykładowa różnica jest taka, że IO Streamy pozwalają nam tylko odczytywać i zapisywać dane (np. z pliku). IO Streamy nie mają wbudowanego mechanizmu do przetwarzania tych danych, jak "functional" Stream i metody `map()` lub `filter()`.

## To czym są te Streamy tutaj?

Zawartość plików może być odczytywana lub zapisywana przy wykorzystaniu IO Streamów (In/Out).

Streamy są strumieniem danych, z którego możesz odczytywać lub do którego możesz zapisywać informacje.

Stream jest strukturą danych, którą można rozumieć jako stały przepływ informacji. Możemy sobie wyobrazić to jako przepływ wody, gdzie kropla po kropli mamy umieszczone jakieś informacje. Istnieją Streamy, które mogą przetwarzać dane kropla po kropli. Są też takie, które mogą przetwarzać kilka kropli jednocześnie (mają wtedy w nazwie `Buffered`).

Streamy nie mają indeksów jak przykładowo w tablicach, nie możemy również przejść w takim Streamie do przodu ani do tyłu, tak jakbyśmy przykładowo mogli w tablicach.

Streamy są typowo oparte o bajty albo znaki. Streamy, które są oparte na bajtach najczęściej mają w swojej nazwie fragment "...Stream". Niedługo poznamy `InputStream` oraz `OutputStream`. Streamy tego typu zapisują oraz odczytują surowe bajty. Streamy oparte o znaki nazywają się najczęściej "...Reader" lub "...Writer".

Pomocne w wyobrażeniu sobie Streamów może być spojrzenie na problem zważenia bardzo ciężkiego worka z piaskiem. Dodajmy do tego, że waga nie jest w stanie zważyć całego worka jednocześnie. Możemy go wtedy podzielić na mniejsze worki przesypując piasek i ważyć je oddzielnie. Czyli dzielimy worek na porcje i procesujemy kolejne porcje.

Pomimo, że wspomnieliśmy, że występują Streamy bajtowe i znakowe, w praktyce pod spodem praktycznie wszystkie Streamy operują na bajtach. Sprowadza się to do odczytu lub zapisu pojedynczych bajtów, ewentualnie tablicy bajtów. Wokół tego zachowania zostały stworzone klasy, które są w stanie zapisywać więcej niż pojedyncze bajty ze względów wydajnościowych. W praktyce dużo czasu zajmuje komunikacja między aplikacją Javową a systemem plików, zatem zapisywanie lub odczytywanie pojedynczych bajtów może się stać niewydajne. Stąd klasy, które starają się zapisywać lub odczytywać wiele bajtów na raz.

Pomimo, że mówimy o Streamach w kontekście odczytu i zapisu plików z dysku, możliwe jest też ich używanie w kontekście komunikacji z innymi serwerami (czyli przez sieć).

# Różnice między Streamami bajtowymi a znakowymi

Jeżeli klasa ma w nazwie Stream, operuje ona wtedy na bajtach. Jeżeli klasa ma w nazwie `Reader` lub `Writer`, oznacza to, że operacje odbywają się na znakach. Pomimo występowania nazewnictwa `Reader` lub `Writer`, nadal są to Streamy.

Pojawia się zatem pytanie, po co stosować Streamy znakowe, skoro można zawsze używać Streamów bajtowych?

Są one dostosowane do typowych operacji tekstowych, zatem ułatwiają operowanie kodem jeżeli chcemy manipulować danymi tekstowymi. Druga rzecz to kodowanie plików. I znowu pojawia się nowy termin, czym jest kodowanie?

Termin "kodowanie plików" lub "kodowanie znaków" nie jest aż taki nowy, pojawiał się już parę razy, ale wyjaśnijmy go sobie raz na zawsze 😀.

## Character encoding

Kodowanie znaków określa w jaki sposób znaki są kodowane i przechowywane w reprezentacji bajtowej i w jaki sposób z bajtów są one dekodowane z powrotem jako znaki. Mamy dostępnych dużo różnych rodzajów kodowań i zwyczajnie określają one jaki znak tekstowy ma być reprezentowany przez podaną wartość bajta.

Jeżeli ktoś byłby zainteresowany zabawą różnymi rodzajami kodowań znaków, to polecam pobawić się poniższym kodem:

```
// Drukujemy na ekranie domyślne kodowanie
System.out.println("Charset.defaultCharset: " + Charset.defaultCharset());

// Zabawa będzie widoczna dopiero na polskich znakach,
// bo to właśnie znaki diaktryczne stanowią najczęściej problem
String sentence = "some sentence to work on ąęół";
byte[] bytes = sentence.getBytes(StandardCharsets.UTF_8);
System.out.println(Arrays.toString(bytes));
String result = new String(bytes, StandardCharsets.UTF_8);
System.out.println(result);
```

Kodowanie UTF-8 jest popularne i sprawdza się w przypadku polskich znaków, ale spróbuj pobawić się np. US_ASCII lub ISO_8859_1. Jeżeli w wymienionych stałych nie ma kodowania, które nas interesuje, możemy wykorzystać metodę:

```
Charset.availableCharsets().keySet().stream().forEach(System.out::println);

// A następnie wykorzystać
Charset.forName("windows-1250");
```

W praktyce warto jest znać podstawowe rodzaje kodowań np. UTF-8 lub UTF-16, natomiast zabawa tym jest potrzebna najczęściej wtedy gdy wystąpi problem ze znakami diaktrycznymi.

## ASCII i Unicode

Omówmy sobie (tak dla przypomnienia) jak działa w Javie typ char.

```
char elementB = 'B';
System.out.println(elementB);
```

Wydrukuje nam B, oczywiście.

Ale możemy użyć też liczby, która reprezentuje dany kod w **ASCII** lub **Unicode**:

```
char elementB = 66;
System.out.println(elementB);
```

Liczba 66 w tablicy znaków ASCII reprezentuje właśnie znak B. Warto wiedzieć, że w tablicy ASCII /Unicode litery wielkie i małe mają przypisane inne liczby, np. małe **b** jest reprezentowane przez liczbę

98.

Procesor pracuje na liczbach, ale my chcemy używać też liter i innych znaków. Jak to zrobić? Właśnie za pomocą kodowania, czyli przypisywania jakiemuś znakowi liczby, która go reprezentuje. Tak powstało ASCII a później Unicode. Unicode jest rozszerzoną wersją ASCII, ASCII bowiem nie zawiera liter i znaków z innych alfabetów. Jeśli chcemy uzyskać taki znak, np małe polskie ł, to musimy posłużyć się liczbą z tablicy Unicode (czyli większym numerem niż daje ASCII). Java i typ char, oczywiście, sobie z tym poradzą. Typ char operuje na tablicy Unicode, zostało to rozwiązane w ten sposób, aby typ ten mógł przechować więcej znaków niż tylko te z tablicy ASCII.

```
char character = 322;
System.out.println(character);
```

zwrócone zostanie ł.

A tutaj znajdziesz linki do tablicy znaków ASCII oraz Unicode.

### Unicode vs UTF-8

Jaka jest zatem różnica między Unicode a UTF-8? Nie jest to jedno i to samo.

Unicode jest standardem, który definiuje w jaki sposób można mapować znaki na numery (numery te są inaczej zwane punktami kodowymi).

UTF-8 jest jednym ze sposobów zamiany punktów kodowych na formę zrozumiałą dla komputera, czyli na bity. Można to rozumieć jak algorytm, który pozwala przekonwertować wspomniane punkty kodowe na sekwencję bitów lub odwrotnie - sekwencję bitów na punkty kodowe. Oprócz UTF-8 istnieje również wiele różnych kodowań.

W pokazywanych przykładach kodu, znaki są zamieniane na tablice byte[], stąd widzimy reprezentację w postaci bajtów, a nie bitów. Dodajmy tutaj też, że czytając o kodowaniu w internecie raz natkniesz się na informacje, że kodowanie znaków to sposób zamiany znaków na bajty, a raz, że na bity. Pamiętając, że 1 bajt to 8 bitów, używane jest to wymiennie.

#### Wróćmy do Streamów

Wracając do Streamów. Powiedzieliśmy, że druga różnica między Reader/Writer, a InputStream /OutputStream to kodowanie plików. Stosując Reader/Writer możemy określić jakie kodowanie znaków nas interesuje, przez co metody będą je stosowały automatycznie. W przypadku InputStream/OutputStream nie mamy takiej możliwości.

Widzimy już, że Streamy znakowe mogą być stosowane do operowania na plikach tekstowych, Streamy bajtowe również mogą być używane do operowania na plikach tekstowych, ale oprócz tego mogą być używane do operowania na obrazkach. Manipulacja obrazkami przy wykorzystaniu Streama tekstowego wydaje się lekko bez sensu ☺.

## Jak możemy sobie podzielić Streamy na kategorie?

Streamy będą miały w nazwach takie słowa kluczowe jak Input albo Output, Reader albo Writer. Na tej podstawie możemy wnioskować, czy służą do zapisu czy do odczytu. Jednocześnie Streamy można

podzielić na takie będące na niskim albo wysokim poziomie. Streamy na niskim poziomie, to takie, które przykładowo czytają bajt po bajcie, podczas gdy te na wysokim poziomie odczytują lub zapisują wiele bajtów jednocześnie. Te na wysokim poziomie mogą mieć w nazwie np. `Buffered`. Streamy na wysokim poziomie mogą przyjmować inne Streamy jako swoje punkty wejściowe.

W końcu przykład kodu:

```
try (ObjectInputStream objectStream = new ObjectInputStream(
        new BufferedInputStream(
            new FileInputStream("someFile.txt")))) {
    // reszta kodu
}
```

Pomimo, że ten zapis może się wydawać dziwny, to mamy tutaj Stream niskiego poziomu `FileInputStream`, który odczytuje plik `someFile.txt` bajt po bajcie. Następnie ten Stream jest opakowany Streamem `BufferedInputStream`, który buforuje odczyt, czyli pozwala na wczytywanie kilku bajtów jednocześnie w celu poprawy wydajności. Całość jest owinięta w `ObjectInputStream`, który pozwoli nam odczytać zawartość pliku jako obiekt.

Zapis jak wyżej jest normalny w tym podejściu, nie jest to jakiś rocket-science.

Trzeba również pamiętać, że nie można zestawiać ze sobą `Readerów` i `Writerów` oraz `Input` i `Output` Streamów, czyli:

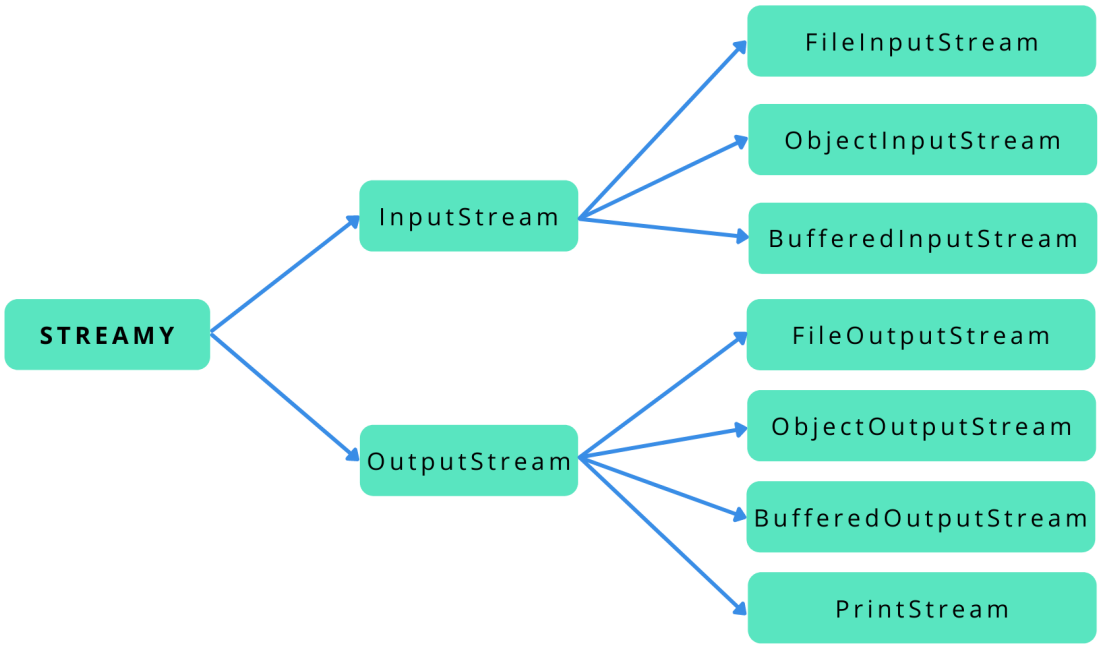
```
// Żaden przykład się nie kompiluje
new BufferedInputStream(new FileWriter("customFile.txt"));
new BufferedWriter(new FileInputStream("customFile.txt"));
new ObjectInputStream(new BufferedOutputStream(new FileOutputStream("customFile.txt")));
```

Poniżej umieszczamy tabelę Streamów, których można używać do operacji opisywanych wcześniej. Nie ucz się jej na pamięć.

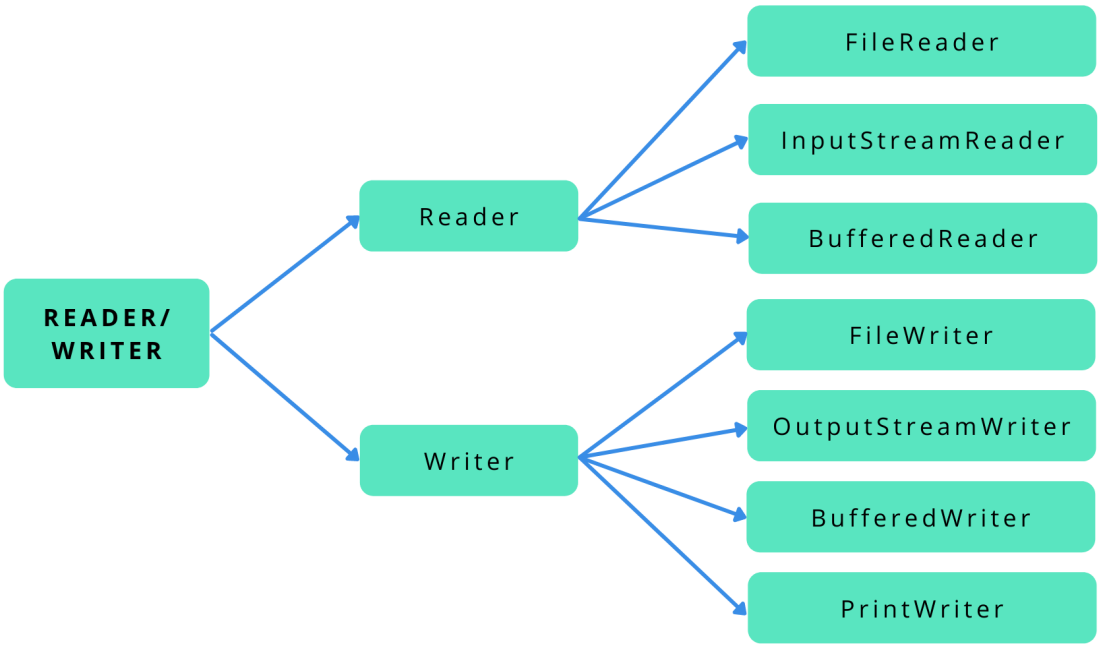
| Nazwa klasy | Opis | Poziom |
|---|---|---|
| Reader | Klasa abstrakcyjna, z której dziedziczą klasy typu Reader | - |
| FileReader | Klasa odczytująca pojedyncze znaki z pliku | Niski |
| InputStreamReader | Klasa odczytująca znaki z podanego InputStreama | Wysoki |
| BufferedReader | Klasa odczytująca znaki z podanego Readera grupując je w celu poprawy wydajności | Wysoki |
| Writer | Klasa abstrakcyjna, z której dziedziczą klasy typu Writer | - |
| FileWriter | Zapisuje dane do pliku jako znaki | Niski |
| OutputStreamWriter | Klasa zapisująca znaki do podanego OutputStreama | Wysoki |
| BufferedWriter | Klasa zapisująca znaki do podanego Writera grupując je w celu poprawy wydajności | Wysoki |

| Nazwa klasy | Opis | Poziom |
|---|---|---|
| PrintWriter | Klasa ułatwiająca formatowanie drukowanych treści | Wysoki |
| InputStream | Klasa abstrakcyjna, z której dziedziczą klasy typu InputStream | - |
| FileInputStream | Klasa odczytująca informacje z pliku w postaci bajtów | Niski |
| BufferedInputStream | Klasa grupująca odczyt z InputStreama w celu poprawy wydajności | Wysoki |
| ObjectInputStream | Klasa będąca w stanie odczytać obiekty Javowe z podanego InputStreama | Wysoki |
| OutputStream | Klasa abstrakcyjna, z której dziedziczą klasy typu OutputStream | - |
| FileOutputStream | Zapisuje dane do pliku w postaci bajtów | Niski |
| BufferedOutputStream | Klasa grupująca zapis do OutputStreama w celu poprawy wydajności | Wysoki |
| ObjectOutputStream | Klasa będąca w stanie zapisać obiekty Javowe do podanego OutputStreama | Wysoki |
| PrintStream | Klasa ułatwiająca formatowanie drukowanych treści | Wysoki |

A gdyby spróbować to samo przedstawić na grafice to wyglądałoby to tak:




*Obraz 1. Hierarchia IO Streamów*




*Obraz 2. Hierarchia Reader/Writer*

# Jak żyć z tymi Streamami?

Teraz przyda nam się konstrukcja `try-with-resources`, która była pokazana wcześniej. Stream jest uważany za resource, czyli może być automatycznie zamknięty w bloku `try-with-resources`. Dzieje się tak bo klasy `InputStream`, `OutputStream`, `Reader` i `Writer` implementują interfejs `Closeable`, który implementuje interfejs `AutoCloseable`.

## Otwieranie i zamykanie

Na tej podstawie wiemy już, że Stream możemy otworzyć, czyli zarezerwować odpowiednie zasoby, aby móc operować na plikach, zapisywać do nich, lub z nich czytać.

## Spuszczanie wody

Kolejnym ciekawym zjawiskiem jest to, że dane, które zapisujemy do Streamów niekoniecznie muszą od razu znaleźć się w pliku, do którego piszemy. Informacje te mogą zostać przetrzymane dłużej w pamięci operacyjnej i dopiero po pewnym czasie zapisane do pliku. Ma to takie konsekwencje, że jeżeli w tym oknie czasowym nasza aplikacja niespodziewanie zakończy działanie, dane te mogą nie zostać zapisane do pliku. Aby takiej sytuacji zapobiec, możemy wykonywać metodę `flush()`, jak spłuczka, możemy ręcznie wypchnąć dane do pliku. Ma to jednak swoją wadę, a mianowicie konsumuje więcej czasu, gdyż operacja faktycznego zapisu do pliku jest kosztowna czasowo. Dlatego powinniśmy jej używać z głową, nie co chwila, tylko przykładowo w odstępach czasu. Jednocześnie należy dodać, że metoda `close()`, która jest wywoływana w bloku `try-with-resources` (pamiętasz, że interfejs `AutoCloseable` definiował taką metodę) również wywołuje `flush()`.

# Pomijanie

Mamy również możliwość pomijania znaków w Streamie i służy do tego metoda `skip()`. Nigdy jej chyba nie użyłem w praktyce...

**W następnej części notatek będzie trochę praktyki (w końcu)...**

# Notatki - Streamy - PrintStream i PrintWriter

## Spis treści



## PrintStream i PrintWriter

`PrintStream` jest klasą, której założeniem jest ułatwienie formatowania danych w `OutputStreamie`, który jest pod spodem. Do tego różni się ona tym, że nie wyrzuca `IOException` jak poprzednio poznane klasy, a w przypadku błędu przestawia wewnętrzną flagę, którą można sprawdzać przy wykorzystaniu metody `checkError()`. Możliwe też jest utworzenie tej klasy w taki sposób aby `flush()` wykonywał się automatycznie po wywołaniu metody drukującej. Klasa ta służy do ułatwienia formatowania drukowanych wartości (np. do pliku albo na konsolę).

`PrintWriter` jest klasą, której założeniem jest ułatwienie formatowania danych w `Writerze`, który jest pod spodem. Zawiera wszystkie metody, które są dostępne w klasie `PrintStream`. Podobnie jak `PrintStream`, dzięki tej klasie możliwe jest ustawienie automatycznego flushowania przy każdym wywołaniu metody `print()`. Podobnie jak `PrintStream`, metody tej klasy nie wyrzucają wyjątków `IOException`, a ewentualne błędy można sprawdzić przez metodę `checkError()`. `PrintWriter` ma również konstruktor, który pozwala zrobić nakładkę `PrintWriter` na `OutputStream`.

Czyli podsumowując, obie te klasy są nakładkami, które dodają nam opcje prostszego formatowania tekstu oraz to, że przy każdym ich wywołaniu nie musimy obsługiwać `IOException`.

Trzymając się jednocześnie konwencji nazewnictwa można wywnioskować, że `PrintStream` zapisuje dane w formie bajtów, podczas gdy `PrintWriter` zapisuje dane przy wykorzystaniu znaków.

I dopiero teraz mogę powiedzieć, że od samego początku nauki używamy `PrintStreama`. Gdzie?

```
System.out.println("some text");
```

Gdy wejdziemy w zmienną `out` w klasie `System`, okaże się, że jest to `PrintStream` drukujący na ekranie. Oprócz zmiennej `out` w klasie `System`, mamy również Stream `err`, który służy do drukowania błędów. Co jednocześnie wyjaśnia, w jaki sposób wyjątki są drukowane w innym kolorze.

```
System.err.println("some error");
```

To teraz wyobraź sobie jak by wyglądało drukowanie elementów na ekranie, gdyby metody `print()` i `println()` deklarowały wyjątek `IOException`.

# in, out oraz err

Czym są zmienne `in`, `out` oraz `err`?

Zmienna `in` reprezentuje standardowe wejście. `Standard Input` jest Streamem, który pozwala nam "komunikować się z aplikacją" i przekazywać jej dane. Najczęściej ten Stream wyłapuje dane wejściowe z klawiatury i przechwytuje w ten sposób wsad od użytkownika. Czyli korzystając z tego Streama możemy wpisywać dane do programu w trakcie jego działania i reagować na wprowadzone w ten sposób dane. Zobaczymy przykład takiego kodu jak przejdziemy do omówienia klasy `Scanner`.

Zmienna `out` reprezentuje standardowe wyjście. `Standard Output` jest Streamem, który pozwala nam drukować dane w terminalu. Jeżeli uruchamiamy aplikację Java w Intellij to tak na prawdę Intellij za nas uruchamia aplikację i pokazuje co jest rezultatem. Możemy natomiast robić to sami w terminalu. Zmienna `out` daje nam możliwość drukowania danych na wyjście standardowe aplikacji - czyli do terminala. W przykładach `PrintStream` będziemy natomiast drukować dane do pliku, a nie na terminal.

Zmienna `err` reprezentuje standardowe wyjścia dla błędów. `Standard Error Output` jest streamem, który służy do drukowania błędów w terminalu. Konwencja mówi, że Stream ten powinien być użyty do wydrukowania informacji, które powinny zostać natychmiast zauważone przez użytkownika aplikacji - czyli np. błędy.

Wspomniane Streamy są otwarte od razu po uruchomieniu aplikacji, nie musimy robić tego ręcznie.

Należy również dodać, że jeżeli będziemy starali się drukować zaraz po sobie `System.out.println()` i `System.err.println()` to nie mamy gwarancji w jakiej kolejności zostaną one wydrukowane. Są to dwa oddzielne Streamy, które drukują informacje na ekranie. Można sobie to wyobrazić w ten sposób, że żaden nie komunikuje się z drugim kiedy wydrukować coś na ekranie żeby zgadzała się kolejność, tylko drukują kiedy im wygodnie 😀.

Możliwe jest ustawienie wspomnianych Streamów `in`, `out` tak żeby operowały na plikach, a nie w terminalu. Domyślnie natomiast, bez żadnych wprowadzonych przez nas zmian operują one na terminalu.

# Metody warte uwagi

Zarówno `PrintStream` i `PrintWriter` ułatwiają drukowanie danych poprzez poniższe metody.

## print()

Najbardziej podstawowa metoda drukująca. Zwróć uwagę, że gdy wykorzystywaliśmy `FileReader` lub `FileWriter` to mogliśmy do metod zapisujących przekazać `String` albo tablicę `char[]`. Metoda `print()` ma wiele przeładowanych swoich wersji, dzięki czemu możemy do niej przekazać dowolny obiekt lub prymityw. Pod spodem zostanie wywołana wtedy metoda `String.valueOf()` i metoda `write()` tak jak

widzieliśmy to wcześniej.

## println()

Metoda `println()` robi to samo co poprzednia, tylko, że po swoim wywołaniu dodaje przejście do nowej linijki. Drugim sposobem na dodanie nowej linijki w `Stringu` jest użycie `\n`. Należy jednak pamiętać o tym, że różne systemy operacyjne dokonują przejścia do nowej linijki na różne sposoby, dlatego lepiej jest używać metody `println()`, bo za nas zostaną rozwiązanie sytuacje skrajne, które mogą doprowadzić do niespodziewanych błędów.

## printf() i format()

Pamiętasz metodę `String.format()`? Te dwie metody (`printf()` i `format()`) robią to samo, tylko, że od razu drukując na ekranie.

**Przejdźmy do kodu:**

```
public class PrintStreamWriterExamples {

    public static void main(String[] args) throws IOException {
        File file = new File("example.txt");
        try (PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter(file)))) {
            writer.print(1L);
            writer.write(String.valueOf(1L));

            Car car = new Car("Roman");
            writer.print(car);
            writer.write(car == null ? "null" : car.toString());

            writer.println(); ①
            writer.println("some stuff"); ②
            writer.printf("some value: [%s]", 5); ③
        }
    }
}
```

① wydrukuje tylko nową linijkę

② wydrukuje "some stuff" i nową linijkę

③ przy tej metodzie trzeba uważać, bo samoistnie nie przechodzi ona do nowej linijki

# Podsumowanie

Chyba właśnie omówione zostały wszystkie bardziej przydatne klasy typu `Stream`, `Reader` oraz `Writer`, które zostały wpisane w tabelce w notatkach ☺. Oprócz poruszonych klas istnieją jeszcze inne, ale nie poruszam ich tutaj, żeby nie było tego za dużo. I tak w praktyce się okaże, że jak będą one potrzebne to będziesz tego szukać w internecie, najprawdopodobniej ☺.

# Notatki - Scanner

## Spis treści



## Scanner

Cały czas rozmawialiśmy o zapisywaniu danych do plików. Wspomnieliśmy też sobie w międzyczasie o `System.in`, czyli Streamie, który pozwala na odczyt danych wprowadzanych przez użytkownika.

Od razu też uprzedzę, że w pracy komercyjnej jest mała szansa, że ta klasa będzie Ci potrzebna, ale warto wiedzieć że coś takiego jest. Czasem może się przydać w testowaniu.

Powiedzieliśmy, że aby wydrukować na ekranie informacje podczas działania aplikacji mamy dostępne 2 `PrintStreamy`: `System.out` i `System.err`. Oprócz nich jest jeszcze `InputStream` - `System.in`. `System.in` jest "surowym" `InputStreamem`. Możemy natomiast uprościć jego używanie stosując klasę `java.util.Scanner`, która opakowuje Stream `System.in` w taki sposób, żebyśmy mogli odwoływać się do całych wprowadzanych w terminalu linijek tekstu.

**Przykład wykorzystania klasy `Scanner`:**

```
public class ScannerExample {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter some data");

        while (scanner.hasNext()) {
            String line = scanner.nextLine();
            System.out.println("Entered: " + line);

            if ("done".equals(line)) {
                break;
            }
        }
    }
}
```

W przykładzie powyżej możemy wprowadzać dane do programu do momentu aż wpiszemy `done`.

## Scanner vs BufferedReader

Znamy już na tym etapie możliwość czytania danych za pomocą klas "...Reader" oraz za pomocą klasy `Scanner`. Jaka jest zatem różnica? Kiedy używać której z nich?

Gdy zaczniemy o tym czytać w internecie to natkniemy się na takie rozróżnienie, że `Scanner` służy do

parsowania, a "...Reader" do czytania. No dobrze, to jaka jest różnica między jednym, a drugim.

- parsowanie - odczyt danych z jakiegoś miejsca z jednoczesną próbą zrozumienia co te dane oznaczają w jakimś kontekście. Czyli czytamy próbując jednocześnie wyjąć z tych danych pewne informacje, które mogą być dla nas znaczące. Prostym przykładem jest fragment tekstu reprezentujący zestaw tagów HTML. Z jednej strony jest to zwykły tekst, ale parsując go możemy przedstawić ten sam tekst jako ustrukturyzowany zestaw tagów HTML.
- czytanie - jak sama nazwa mówi, program może czytać dane bez jednoczesnej próby interpretacji ich znaczenia.

Jak się to ma to klas `Scanner` oraz np. `BufferedReader`? Klasa `Scanner` daje nam metody takie jak np. `nextInt()` lub `nextBigDecimal()`, które świadczą o tym, że przy odczycie próbujemy od razu konwertować te dane na jakiś kontekst, który jet dla nas zrozumiały. Dla odmiany `BufferedReader` nie ma takich metod, służy on tylko do czytania danych, bez próby ich rozumienia w jakimś kontekście.

# Notatki - NIO.2 Paths

# Spis treści



# NIO.2, Path i Paths

I nie jest to przekręcona nazwa (NIO.2) - tak się to nazywa.

Może zaczynając trochę od historii zmian w API Javy w kontekście klas służących do operowania na plikach.

Paczka `java.io` została wprowadzona w Javie w wersji 1.0, czyli dawno temu. Były tam zawarte klasy umożliwiające operowanie na plikach przy wykorzystaniu pojedynczych bajtów, czyli klasy z nazwami Stream, których tematykę poruszaliśmy.

W Javie w wersji 1.4 zostały wprowadzone m.in. taki klasy jak `ByteBuffer` lub `Channels`, które z założenia miały poprawiać wydajność poprzednich rozwiązań. Klasy wprowadzone w wersji 1.4 wyróżniały się tym, że stosowały rozwiązania nieblokujące (non-blocking). Oznacza to, że poprzednie rozwiązania blokowały zasób na którym pracowały, czyli przykładowo plik na którym operowały. Od teraz możliwe było stosowanie tymczasowego bufora, który nie blokował zasobu, na którym pracujemy. Zbiorczo te rozwiązania są określane jako NIO (non-blocking IO) i dodane zostały w paczce `java.nio`. Nie

poruszaliśmy natomiast tematyki ich wykorzystania.

W Javie 7 zostały wprowadzone kolejne klasy, które określa się zbiorczo jako **NIO.2**. Gdzieś kiedyś przeczytałem taką ciekawostkę, że zmiany w Javie 1.4 miały zastąpić rozwiązania wprowadzone na samym początku, natomiast dopiero zmiany wprowadzone w Javie 7 faktycznie to zrobiły ☺.

W tej notatce poruszymy tematykę klas wprowadzonych w Javie 7. Dołożymy też do tego zmiany wprowadzone przez Javę 8, doszły wtedy metody umożliwiające funkcyjne podejście do tematu. Jak też z każdymi poprawkami, te poprawki również miały poprawić wydajność działania aplikacji ☺.

Jeżeli się zastanawiasz czemu poruszaliśmy tematykę Streamów, skoro teraz napisałem, że w Javie 7 wprowadzono lepsze rzeczy? Bo w praktyce nadal często można spotkać użycie wspomnianych Streamów. A druga kwestia to jest rozumienie sposobu działania pod spodem. Niedługo zostaną pokazane inne sposoby, których będziesz prawdopodobnie używać najczęściej, natomiast warto jest znać sposób działania mechanizmów "pod spodem". Często będzie to wyglądało w ten sam sposób w odniesieniu do innych zagadnień, czyli będziemy poruszać przykłady kodu, których nie będziesz praktycznie pisać w praktyce, ale bardzo ważne jest w takich przypadkach zrozumienie sposobu działania mechanizmu. Ważne jest to z tego powodu, że często mechanizmy działają do siebie analogicznie lub podobnie.

Dlaczego nie wspominam rzeczach wprowadzonych np. w Java 15, a skupiam się tylko wokół rzeczy wprowadzonych w Java 7 lub 8? Bo w nowszych wersjach nie zostały wprowadzone zmiany dotyczące tych zagadnień, które byłby istotne na tym etapie nauki.

# Path

Do tych wszystkich klas, które już poznaliśmy dorzucimy jeszcze interfejs `java.nio.Path`. Można to rozumieć jako byt reprezentujący ścieżki w systemie operacyjnym razem z całą hierarchią tych ścieżek. Ścieżka taka może wskazywać na plik lub folder. Rozumienie było takie, aby interfejs `Path` zastąpił klasę `File`, którą poznaliśmy wcześniej. Wspominamy o obu z nich, bo obydwie spotyka się ciągle w użyciu.

Filozoficznie patrząc, interfejs `Path` może być używane zamiast `File` gdyż dostarcza on tych samych funkcjonalności, dodaje on również nową funkcjonalność. Wspiera dowiązania symboliczne (*symbolic links*, *symlink*). Są to takie pliki, które zawierają odniesienia do innego pliku lub katalogu w postaci ścieżki do tego pliku lub katalogu. Po co się to stosuje? Jeżeli przykładowo chcemy mieć w projekcie odniesienie do jakiegoś pliku, który może często zmieniać lokalizację. W takim przypadku możemy w konfiguracji projektu dodać lokalizację takiego symlinka i w symlinku zmieniać namiar na plik, który nas interesuje. Na ten moment żeby dalej nie komplikować, nie będziemy zagłębiać się w temat.

## Jak stworzyć Path

Tutaj zaczyna się zabawa. W przypadku klasy `File`, wywoływaliśmy konstruktor podając namiar na plik na dysku i wszystko działało. Tutaj już tak nie jest ☺. Pierwsza rzecz jest taka, że `Path` jest interfejsem, więc nie ma konstruktora, żeby stworzyć instancję. Pojawia się zatem pytanie, czemu `Path` jest interfejsem. W nowym podejściu (czyli tym NIO.2), utworzenie w Javie obiektu reprezentującego folder lub plik jest operacją zależną od systemu plików, co wynika z systemu operacyjnego, na którym pracujemy. Czyli `Path` jest tylko interfejsem, który zapewnia nam dostęp do metod, natomiast konkretna

implementacja zależna od systemu plików musi nam zostać dostarczona w inny sposób.

I wtedy na białym koniu wjeżdża klasa `java.nio.files.Paths`, która ma zdefiniowane metody statyczne, pozwalające na utworzenie obiektu implementującego interfejs `Path`. Jednocześnie robi to w taki sposób, że logika pod spodem jest zależna od systemu plików, jaki funkcjonuje w danym systemie operacyjnym. Natomiast z perspektywy programisty piszącego kod, możemy używać metod z klasy `Paths` i dla nas jest to przeźroczyste.

**Przykład użycia w kodzie:**

```
Path path1 = Paths.get("src/pl/zajavka/java/nio/myFile.txt");

// Druga opcja jest o tyle dobra, że nie musimy się zastanawiać nad separatorem
Path path2 = Paths.get("src", "pl", "zajavka", "java", "nio", "myFile.txt");
Path path3 = Paths.get("nio", "myFile.txt");

// Nie ma metody path.exists() tak jak to było w file
System.out.println("path1: " + path1 + " exists: " + Files.exists(path1));
System.out.println("path2: " + path2 + " exists: " + Files.exists(path2));
System.out.println("path3: " + path3 + " exists: " + Files.exists(path3));
```

Już widzisz, że mamy takie klasy/interfejsy jak `Path`, `Paths`, `File`, `Files`. O klasie `Files` napiszemy później. Można natomiast powiedzieć, że `Paths` i `Files` są takimi klasami 'pomocniczymi', które zawierają metody statyczne aby coś zrobić. Tak jak w tym przypadku, przy wykorzytaniu klasy `Files` nastąpiło sprawdzenie, czy plik pod podaną ścieżką istnieje.

# URI

Skoro mówimy o `Path`, należy też wspomnieć o `java.net.URI`. Jedna z metod `Paths.get()` przyjmuje obiekt `URI` jako argument. `URI` (*Uniform Resource Identifier*) jest ustandaryzowanym sposobem na identyfikację jakiegoś zasobu. Jego zapis rozpoczyna się od schematu, przykładowo `http://`, `https://`, `file://`, a następnie zawiera lokalizację zasobu.

Wiedząc już o jego istnieniu, można napisać kod pokazany poprzednio w taki sposób:

```
try {
    Path path1 = Paths.get(new URI("src/pl/zajavka/java/nio/myFile.txt"));
} catch (Exception e) {
    System.err.println("path1 missing scheme: " + e.getMessage());
}
```

Trzeba natomiast pamiętać, że używając `URI` należy podać schemat zasobu, zatem przykład wyżej jest nieprawidłowy. Poniżej jest prawidłowy.

```
try {
    Path path2 = Paths.get(new URI("file://src/pl/zajavka/java/nio/myFile.txt"));
} catch (Exception e) {
    System.out.println("won't print here");
}
```

# FileSystem

Dostępna jest również klasa `java.nio.file.FileSystem`, która jest w stanie pokazać nam jaki obiekt systemu plików faktycznie jest tworzony na naszej maszynie. Możemy napisać taki kod:

```
FileSystem fileSystem = FileSystems.getDefault();
System.out.println(fileSystem);
```

W moim przypadku drukuje się coś podobnego do:

```
sun.nio.fs.WindowsFileSystem@3a03464
```

Korzystając teraz ze zmiennej `fileSystem` możemy wywołać:

```
Path path = fileSystem.getPath("src/pl/zajavka/java/nio/myFile.txt");
```

Co da nam ten sam efekt, co wywołanie `Paths.get()`.

# Konwersja między Path a File

Jeżeli natomiast chcielibyśmy przejść z używania klasy `Path` na `File` i odwrotnie, można to zrobić w ten sposób:

```
File file1 = new File("someFile.txt");
Path path1 = file1.toPath();

Path path2 = Paths.get("someFile.txt");
File file2 = path2.toFile();
```

# Operacje na obiektach typu Path

Tak sobie piszemy o tym interfejsie `Path` i nie napisaliśmy jednej najważniejszej rzeczy, o której trzeba pamiętać. Jeżeli stworzymy obiekt `Path`, to nie jest to plik, który istnieje na dysku, tylko reprezentacja lokalizacji pliku w systemie operacyjnym. Taki plik wcale nie musi fizycznie istnieć na dysku, żeby móc operować na obiekcie `Path`. `Path` dostarcza nam natomiast metody, które wymagają aby plik faktycznie się na tym dysku znajdował, inaczej dostaniemy wyjątek.

Przejdźmy zatem do omówienia kolejnych metod.

## toString()

Zwraca reprezentację `Path` jako `String`. Standardowa metoda z klasy `Object`. Jest to jedna z niewielu metod z Path, która zwraca `Stringa`. Uczulam na to, bo inne metody zwracają `Path`.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
String toString = path.toString();
System.out.println(toString); ①
```

① Wydrukowane zostanie: C:\zajavka\someFile.txt

## getNameCount()

Ta metoda jest używana aby otrzymać ilość "elementów" w ścieżce, czyli w praktyce zlicza nam ilość folderów w podanej ścieżce razem z plikiem, który może wystąpić na końcu ścieżki.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
int nameCount = path.getNameCount();
System.out.println(nameCount); ①
```

① Wydrukowane zostanie: 2

## getName(int index)

Ta metoda zwraca nam nowy obiekt typu `Path`, który występuje na miejscu podanego indeksu w oryginalnym obiekcie `Path`. Warto zwrócić uwagę, że indeksowanie jest od `0`. Kolejna ważna rzecz, to to, że `path.getName(0)` nie zwraca nam roota, czyli w przypadku np. Windowsa, nie dostaniemy np. Dysku `C`, tylko pierwszy katalog na ścieżce.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path name0 = path.getName(0);
System.out.println(name0); ①
```

① Wydrukowane zostanie: zajavka

## getFileName()

Metoda zwraca nam nazwę pliku, który jest wskazany na ścieżce. Plik jest rozumiany jako najdalszy element od początku ścieżki, czyli w przypadku braku zdefiniowanego pliku w ścieżce (gdy ścieżka zawiera same katalogi), metoda ta zwróci ostatni katalog.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path fileName = path.getFileName();
System.out.println(fileName); ①

Path path1 = Paths.get("C:/zajavka/someCatalog");
Path fileName1 = path1.getFileName();
System.out.println(fileName1); ②
```

① Wydrukowane zostanie: someFile.txt

② Wydrukowane zostanie: someCatalog

## toAbsolutePath()

Metoda zwraca nam obiekt typu `Path` reprezentujący absolutną ścieżkę utworzoną na podstawie ścieżki, którą podaliśmy. Jeżeli podajemy ścieżkę relatywną, wywołanie `toAbsolutePath()` stworzy nam ścieżkę absolutną na podstawie miejsca, w którym uruchamiany jest projekt w systemie operacyjnym.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
Path absolutePath = path.toAbsolutePath();
System.out.println(absolutePath); ①

Path path1 = Paths.get("zajavka/someFile.txt");
Path absolutePath1 = path1.toAbsolutePath();
System.out.println(absolutePath1); ②
```

① Wydrukowane zostanie: C:\zajavka\someFile.txt

② Wydrukowane zostanie: C:\Users\krogowski\zajavka\someFile.txt

## isAbsolute()

Metoda zwraca `boolean` mówiący czy podana ścieżka jest absolutna czy nie.

```
Path path = Paths.get("C:/zajavka/someFile.txt");
boolean absolute = path.isAbsolute();
System.out.println(absolute); ①

Path path1 = Paths.get("zajavka/someFile.txt");
boolean absolute1 = path1.isAbsolute();
System.out.println(absolute1); ②
```

① Wydrukowane zostanie: true

② Wydrukowane zostanie: false

## getParent()

Metoda zwraca `Path` reprezentujący rodzica ostatniego pliku lub katalogu na ścieżce. Przykładowo:

```
Path path = Paths.get("C:/zajavka/someDirectory/someFile.txt");
Path parent = path.getParent();
System.out.println(parent); ①

Path path1 = Paths.get("zajavka/someDirectory/someFile.txt");
Path parent1 = path1.getParent();
System.out.println(parent1); ②
```

① Wydrukowane zostanie: C:\zajavka\someDirectory

② Wydrukowane zostanie: zajavka\someDirectory

## getRoot()

Pobierz korzeń... Metoda ta zwraca katalog główny (katalog root), czyli katalog nadrzędny dla

wszystkich innych katalogów. Jeżeli ścieżka jest relatywna, zostanie zwrócony `null`.

```
Path path = Paths.get("C:/zajavka/someDirectory/someFile.txt");
Path root = path.getRoot();
System.out.println(root); ①

Path path1 = Paths.get("zajavka/someDirectory/someFile.txt");
Path root1 = path1.getRoot();
System.out.println(root1); ②
```

① Wydrukowane zostanie: C:\

② Wydrukowane zostanie: null

## subpath()

Metoda ta analogicznie jak `substring()` służy do stworzenia podścieżki na podstawie przekazanych indeksów ścieżki oryginalnej. Pierwszy argument jest inkluzywny, drugi nieinkluzywny (ekskluzywny, znaczy taki premium? 😄).

```
Path path = Paths.get("C:/zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path subpath1 = path.subpath(1, 3);
Path subpath2 = path.subpath(2, 4);
Path subpath3 = path.subpath(0, 2);
System.out.println(subpath1); ①
System.out.println(subpath2); ②
System.out.println(subpath3); ③

Path path1 = Paths.get("zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path subpath4 = path1.subpath(1, 3);
Path subpath5 = path1.subpath(2, 4);
Path subpath6 = path1.subpath(0, 2);
System.out.println(subpath4); ④
System.out.println(subpath5); ⑤
System.out.println(subpath6); ⑥
```

① Wydrukowane zostanie: someDirectory\anotherDirectory

② Wydrukowane zostanie: anotherDirectory\yetAnother

③ Wydrukowane zostanie: zajavka\someDirectory

④ Wydrukowane zostanie: someDirectory\anotherDirectory

⑤ Wydrukowane zostanie: anotherDirectory\yetAnother

⑥ Wydrukowane zostanie: zajavka\someDirectory

## relativize()

Metoda ta służy do skonstruowania relatywnej ścieżki ze ścieżki pierwszej do ścieżki drugiej. Innymi słowy, efektem jest ścieżka, po której przejściu ze ścieżki pierwszej, znajdziemy się w ścieżce drugiej.

```
Path path1 = Paths.get("E:/zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
Path path2 = Paths.get("E:/zajavka/someDirectory/anotherDirectory/secondDirectory/anotherFile.txt");
System.out.println(path1.relativize(path2)); ①
System.out.println(path2.relativize(path1)); ②

Path path3 = Paths.get("someFile.txt");
Path path4 = Paths.get("anotherFile.txt");
System.out.println(path3.relativize(path4)); ③
System.out.println(path4.relativize(path3)); ④
```

① Wydrukowane zostanie: ..\..\secondDirectory\anotherFile.txt

② Wydrukowane zostanie: ..\..\yetAnother\someFile.txt

③ Wydrukowane zostanie: ..\anotherFile.txt

④ Wydrukowane zostanie: ..\someFile.txt

Przykładach `path3` i `path4` pojawiają się `..` gdyż aby przejść z pliku `someFile.txt` do `anotherFile.txt` musimy najpierw dostać się do katalogu rodzica. W odwrotną stronę działa to identycznie.

Jeżeli mamy do czynienia z 2 ścieżkami absolutnymi jak w podanym przypadku, Java nie wykonuje sprawdzenia czy ścieżki te faktycznie istnieją na dysku, stąd też podałem ścieżki na dysku `E`, który na mojej maszynie nie istnieje.

## resolve()

A może chcemy dodać do siebie 2 ścieżki? To też tak można.

```
Path path5 = Paths.get("zajavka/../someDirectory/anotherDirectory");
Path path6 = Paths.get("secondDirectory/../anotherCatalog");
System.out.println(path5.resolve(path6)); ①
```

① Wydrukowane zostanie: zajavka\.\someDirectory\anotherDirectory\secondDirectory\..\anotherCatalog_

Trzeba tutaj natomiast zwrócić uwagę, że ta metoda "nie sprząta" ścieżek. Tzn, że jeżeli na ścieżce występują znaki `..`, to ta metoda tego nie sprzątnie, tylko przeklei dalej tak jak było. Od sprzątania jest następna metoda.

## normalize()

Tą metodą możemy posprzątać naszą ścieżkę. Czyli jeżeli na naszej ścieżce występowały `..`, to `normalize()` skróci tę ścieżkę w taki sposób, aby wynik końcowy był taki sam jak przed wywołaniem tej metody, ale zapis będzie krótszy.

```
Path path7 = Paths.get("zajavka/../someDirectory/anotherDirectory/secondDirectory/../anotherCatalog");
System.out.println(path7.normalize()); ①
```

① Wydrukowane zostanie: someDirectory\anotherDirectory\anotherCatalog

Tutaj znowu należy zaznaczyć, że Java nie sprawdza czy podana ścieżka faktycznie występuje na dysku.

### toRealPath()

I ostatnia metoda jaką poruszaliśmy. Ta metoda zachowuje się trochę jak `toAbsolutePath()`, przy czym sprawdza jednocześnie czy plik faktycznie istnieje na dysku. Jeżeli nie istnieje, to wyrzuca znany nam już `NoSuchFileException`.

Warto też wiedzieć, że jeżeli mamy zdefiniowaną naszą ścieżkę jako nieznormalizowaną (czyli np. z `..`) to ta metoda pod spodem dokona normalizacji takiej ścieżki.

```
Path path1 = Paths.get("zajavka/someDirectory/anotherDirectory/yetAnother/someFile.txt");
try {
    System.out.println(path1.toRealPath());
} catch (IOException e) { ①
    e.printStackTrace();
}
```

① Jeżeli plik nie istnieje, zostanie wyrzucone `java.nio.file.NoSuchFileException`

# Working directory

I na koniec, znając już te wszystkie rzeczy, warto jest dodać dosyć ważną kwestię. Koncepcję czegoś takiego jak **working directory** - które jest używane gdy pracujemy na ścieżkach relatywnych. Jeżeli w naszej aplikacji będziemy podawać ścieżkę relatywną, to będzie ona użyta w kontekście naszego **working directory**, czyli jakby katalogu w którym odbywa się cała praca. Jak ustalić namiar na taki katalog?

Jeżeli spróbujemy odpalić przykłady takie jak np:

```
Path relative1 = Paths.get("java/nio_path/myFile.txt");

try {
    System.out.println("relative1.toRealPath(): " + relative1.toRealPath());
} catch (IOException e) {
    System.err.println("relative1 error: " + e.getMessage());
}
```

lub

```
Path relative1 = Paths.get("src/java/nio_path/myFile.txt");

try {
    System.out.println("relative1.toRealPath(): " + relative1.toRealPath());
} catch (IOException e) {
    System.err.println("relative1 error: " + e.getMessage());
}
```

To zobaczymy, że wywołując `toRealPath()`, Java zawsze postara się naszą ścieżkę relatywną dokleić do obecnego **working directory** (dodam, że każda aplikacja będzie miała swoje **WD**) i w konsekwencji wydrukowane zostanie coś takiego:

```
relative1 error: src/java/nio_path/myFile.txt
```

Przy czym moim **WD** jest:

```
C:\Users\krogowski\zajavka
```

Czyli Java próbuje naszą ścieżkę relatywną rozwiązać w kontekście swojego **WD**. Jak natomiast można sprawdzić jakie mamy **WD**?

```
try {
    System.out.println("Current WD: " + Paths.get(".").toRealPath()); ①
} catch (IOException e) {
    System.err.println("relative3 error: " + e.getMessage());
}
```

① Wydrukowane zostanie: Current WD: C:\Users\krogowski\zajavka

# Notatki - NIO.2 Files

# Spis treści



# NIO.2 i Files

No dobrze, umiemy się już poruszać po ścieżkach. Teraz możemy przejść do omówienia dostępnych klas "po nowemu" w kontekście faktycznego dostępu do plików i operowania na nich.

W tym celu wprowadzimy sobie klasę `java.nio.file.Files`. Tutaj zwracam uwagę na to podobieństwo między `Path` i `Paths`, a także `File` i `Files`. `Paths` i `Files` są klasami pomocniczymi, które będziemy wykorzystywać aby mieć dostęp do ich metod statycznych. `Path` i `File` są klasami, które reprezentują ścieżkę oraz plik na dysku.

Klasa `Files` dostarcza nam metody, które już poznaliśmy. Metody te pozwalają nam wykonać operacje takie jak przeniesienie pliku między katalogami, albo zmiana jego nazwy. Dlatego też najpierw poruszymy te metody, a potem zestawimy je sobie w tabelce z tymi, które już poznaliśmy w klasie `File`.

# exists()

Najprostsza metoda, która pozwala sprawdzić, czy podana ścieżka faktycznie może zostać znaleziona na dysku. Możemy za jej pomocą sprawdzić występowanie zarówno pliku jak i katalogu. Metoda zwraca `boolean`.

```
Path path = Paths.get("zajavka/myFile.txt");
System.out.println(Files.exists(path)); ①
```

① Wydrukowane zostanie: `true` albo `false`.

# createDirectory()

Jak sama nazwa wskazuje, metoda ta służy do stworzenia katalogu. Jeżeli chcemy stworzyć po drodze kilka katalogów, bo przykładowo połowa ścieżki nie istnieje - do tego mamy metodę poniżej, czyli `createDirectories()`. Typem zwracanym z metody jest `Path`.

```
Path createDirectory = Paths.get("zajavka/newDir");
try {
    Files.createDirectory(createDirectory); ①
} catch (IOException e) {
    System.err.println("Files.createDirectory(createDirectory): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie katalogu `newDir` pod warunkiem, że katalog `zajavka` istniał już wcześniej.

# createDirectories()

Jeżeli po drodze występuje kilka katalogów, które nie istnieją, a my chcemy stworzyć wszystkie po drodze do tego ostatniego, który nas interesuje to poprzednia metoda nie odniesie efektu (zostanie wyrzucony wyjątek). Należy wtedy wykorzystać `createDirectories()`. Typem zwracanym z metody jest `Path`.

```
Path createDirectories = Paths.get("zajavka/newDir/newDir2/newDir3");
try {
    Files.createDirectories(createDirectories); ①
} catch (IOException e) {
    System.err.println("Files.createDirectories(createDirectory): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie katalogu katalogów wymienionych na ścieżce przypisanej do zmiennej `createDirectories`.

# copy()

Ta metoda jak sama nazwa wskazuje służy do kopiowania ☺. Typem zwracanym z metody jest `Path`. Przykład kopiowania pliku.

```
Path oldPath = Paths.get("zajavka/myFile.txt");
Path newPath = Paths.get("zajavka/myNewFile.txt");
try {
    Files.copy(oldPath, newPath); ①
} catch (IOException e) {
    System.err.println("Files.copy(oldPath, newPath): " + e.getMessage());
}
```

① Wynikiem działania metody będzie stworzenie kopii pliku określonej pod ścieżką `newPath`.

A co z kopiowaniem katalogów? Jeżeli chcemy kopiować katalogi, to nastąpi tak zwane "shallow copy", czyli zawartość katalogu nie zostanie skopiowana, a tylko sam katalog.

W takim przypadku musimy sami zatroszczyć się o kopiowanie odpowiednich plików. Domyślnie też, jeżeli plik, który kopiujemy istnieje już w miejscu docelowym na dysku, nie zostanie on nadpisany.

Trzeba to wymusić przez podanie odpowiednich parametrów do metody. Nie chcę natomiast schodzić na tyle głęboko w temat, bo jak będzie to potrzebne w pracy to i tak będziesz to Googlować ☺.

# move()

Jak sama nazwa wskazuje, metoda ta służy do przeprowadzki pliku. Generalnie to jest o tyle sprytne, że za pomocą tej metody można również zmienić nazwę pliku, bo przecież filozoficznie rzecz biorąc, możemy przenieść plik do tego samego katalogu, ale pod inną nazwą pliku i też będzie to move ☺.

Tak samo jak w poprzednim przykładzie jeżeli plik, który przenosimy istnieje już w miejscu docelowym na dysku, nie zostanie on nadpisany. Trzeba to wymusić przez podanie odpowiednich parametrów do metody. Nie chcę natomiast schodzić na tyle głęboko w temat, bo jak będzie to potrzebne w pracy to i tak będziesz to googlować ☺. Typem zwracanym z metody jest `Path`.

```
Path oldPath = Paths.get("zajavka/myFile.txt");
Path newPath = Paths.get("zajavka/myDir/myNewFile.txt");
try {
    Files.move(oldPath, newPath);
} catch (IOException e) {
    System.err.println("Files.move(oldPath, newPath): " + e.getMessage());
}

Path oldPath1 = Paths.get("zajavka/myDir");
Path newPath1 = Paths.get("zajavka/myNewDir");
try {
    Files.move(oldPath1, newPath1);
} catch (IOException e) {
    System.err.println("Files.move(oldPath1, newPath1): " + e.getMessage());
}
```

Wynikiem działania powyższych przykładów będzie przeniesienie plików/katalogów ze starych ścieżek na nowe ścieżki.

# delete i deleteIfExists()

Raczej nie będzie to zaskoczenie, jak napiszę, że ta metoda usuwa zasób znajdujący się pod podaną ścieżką. Ale... trzeba pamiętać, że występuje pare przypadków, w których takie usunięcie nie może zostać wykonane.

Żeby nie uczyć się wszystkich na pamięć, najważniejsze jest zapamiętanie, że jeżeli ścieżka reprezentuje katalog, w którym znajdują się pliki - wywołanie metody wyrzuci wyjątek typu `IOException`.

Do tego przykładowo, gdy ścieżka, którą chcemy usunąć nie istnieje - również zostanie wyrzucony wyjątek, chyba, że użyjemy metody `deleteIfExists()`. Metoda `deleteIfExists()` nie wyrzuca wyjątku, gdy zasób który chcemy usunąć nie istnieje. Zamiast tego zwróci `boolean` sugerujący czy usunięcie się powiodło czy nie.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka/myDir");

try {
    Files.delete(path1);
} catch (IOException e) {
    System.err.println("Files.delete(path1): " + e.getMessage());
}

try {
    Files.delete(path2);
} catch (IOException e) {
    System.err.println("Files.delete(path2): " + e.getMessage());
}
```

Metoda `delete()` nic nie zwraca, a efektem jej działania jest usunięcie podanej ścieżki, pod warunkiem, że ścieżka istnieje.

A jeżeli mamy przypadek próby usunięcia zasobu, którego faktycznie nie ma:

```
Path path3 = Paths.get("zajavka/nonExisting.txt");

try {
    Files.deleteIfExists(path3);
} catch (IOException e) {
    System.err.println("Files.deleteIfExists(path3): " + e.getMessage());
}
```

# readAllLines()

Tak jak mówi nazwa metody, dzięki niej możemy odczytać wszystkie linijki z pliku. Natomiast trzeba z tą metodą uważać, bo wszystkie linijki z pliku są odczytywane na raz i zapisywane do listy Stringów.

Jeżeli plik jest wystarczająco duży, możemy dostać `OutOfMemoryError`. Dalej będzie pokazana metoda, która pozwala poradzić sobie z takimi przypadkami. Także `readAllLines()` stosujmy raczej do małych plików.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    List<String> allLines = Files.readAllLines(path); ①
    allLines.forEach(line -> System.out.println("Line: " + line)); ②
} catch (IOException e) {
    System.err.println("Files.readAllLines(path): " + e.getMessage());
}
```

① Wczytanie całej zawartości pliku jako lista Stringów.

② Wydrukowanie każdej linijki z listy na ekranie.

# isSameFile()

Jeżeli chcemy sprawdzić czy 2 ścieżki wskazuję na ten sam zasób w naszym systemie plików, używamy do tego `isSameFile()`. Mówiąc zasób, mam też na myśli katalogi.

W praktyce metoda ta działa tak, że w pierwszej kolejności sprawdza, czy ścieżki są sobie równe w rozumieniu metody `equals()` (nie następuje tutaj żadna weryfikacja czy zasoby istnieją fizycznie na dysku). Jeżeli takie sprawdzenie zwróci `false`, następuje fizyczne sprawdzenie, czy ścieżki na dysku wskazują na ten zasób. Należy tutaj dodać, że metoda `isSameFile()` nie sprawdzana zawartości plików, służy do sprawdzania ścieżek. Jeżeli któryś z plików nie istnieje na dysku, zostaje wyrzucone `IOException`.

Przykład kodu:

```
Path path1 = Paths.get("zajavka/someDirectory/isSameFile.txt");
Path path2 = Paths.get("zajavka/someDirectory/isSameFile.txt");

Path path3 = Paths.get("zajavka/someDirectory/testingDir/../isSameFile2.txt");
Path path4 = Paths.get("zajavka/someDirectory/isSameFile2.txt");

Path path5 = Paths.get("zajavka/./someDirectory/./isSameFile2.txt");
Path path6 = Paths.get("zajavka/someDirectory/isSameFile2.txt");

Path path7 = Paths.get("zajavka/someDirectory/someDirectory1");
Path path8 = Paths.get("zajavka/someDirectory/someDirectory2");

try { ①
    System.out.println(Files.isSameFile(path1, path2));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path1, path2): " + e);
}

try { ②
    System.out.println(Files.isSameFile(path3, path4));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path3, path4): " + e);
}
```

```
try { ③
    System.out.println(Files.isSameFile(path5, path6));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path5, path6): " + e);
}

try { ④
    System.out.println(Files.isSameFile(path7, path8));
} catch (IOException e) {
    System.err.println("Files.isSameFile(path7, path8): " + e);
}
```

① Ścieżki `path1` oraz `path2` nie muszą istnieć na dysku. Na ekranie zostanie wydrukowane: `true`

② Jeżeli plik `isSameFile2.txt` istnieje na dysku - zostanie wydrukowane `true`. Jeżeli nie istnieje - zostanie wyrzucony wyjątek.

③ Jeżeli plik `isSameFile2.txt` istnieje na dysku - zostanie wydrukowane `true`. Jeżeli nie istnieje - zostanie wyrzucony wyjątek.

④ Jeżeli ścieżki `path7` i `path8` istnieją na dysku - zostanie wydrukowane `false`. Jeżeli nie istnieją - zostanie wyrzucony wyjątek.

# New Buffered readers

Kolejne Streamy i znowu pojawia się pytanie po co. Bo z każdą nową wersją, którą wprowadzają twórcy, ich założeniem jest poprawić wydajność rozwiązania i często możliwie skrócić zapis kodu dla danego mechanizmu. Tak też jest w tym przypadku.

```
Path readingPath = Paths.get("/zajavka/bufferedReadingFile.txt");
try (BufferedReader reader = Files.newBufferedReader(readingPath)) {
    String line = null;
    while((line = reader.readLine()) != null) {
        System.out.println("nextLine: " + line); ①
    }
} catch (IOException e) {
    e.printStackTrace();
}

Path writingPath = Paths.get("/zajavka/bufferWritingFile.txt");
List<String> data = new ArrayList<>();
try (BufferedWriter writer = Files.newBufferedWriter(writingPath)) {
    writer.write("zajavka!"); ②
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowana cała zawartość pliku pod ścieżką `readingPath`.

② Do pliku pod ścieżką `writingPath` zostanie zapisana treść `zajavka!`.

# File Attributes

W tej sekcji chciałem poruszyć zagadnienie funkcjonalności, które dostarcza nam klasa `Files` w odniesieniu do atrybutów plików. Wcześniej cały czas pokazywaliśmy sobie metody dotyczące ścieżek, teraz porozmawiamy o atrybutach. Inaczej mówiąc, porozmawiamy o metadanych (czyli danych o danych) w odniesieniu do plików i/lub katalogów. Czyli o informacjach takich jak przykładowo data utworzenia pliku, nie będziemy tutaj poruszać kwestii zawartości pliku. Tutaj należy też dodać, że niektóre systemy operacyjne przechowują informacje o plikach charakterystyczne dla danego systemu operacyjnego. Czyli przykładowo mogą wystąpić atrybuty pliku, które występują w systemie Ubuntu, a nie będzie ich na Windowsie.

## size()

Nazwa jest chyba wyczerpująca ☺. W ten sposób sprawdzamy rozmiar naszego pliku w bajtach. Metoda ta służy do sprawdzania rozmiaru pliku, jeżeli chcemy sprawdzić rozmiar folderu, jest to zależne od tego na jakim systemie operacyjnym pracujemy. Jeżeli nie zadziała nam sprawdzenie rozmiaru folderu przy wykorzystaniu tej metody należy dodać ręcznie do siebie rozmiar wszystkich plików w folderze.

```
try {
    System.out.println(Files.size(Paths.get("zajavka/myFile.txt"))); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowany rozmiar pliku w bajtach.

## getLastModifiedTime() i setLastModifiedTime()

Tutaj należy zwrócić uwagę na to, że metoda `getLastModifiedTime()` zwraca obiekt klasy `FileTime`. Później możemy wyciągnąć z tego informacje za pomocą metod `toString()`, `toMilis()` albo `toInstant()`.

Mamy również możliwość ręcznego ustawienia daty ostatniej modyfikacji pliku poprzez `setLastModifiedTime()`, która przyjmuje obiekt klasy `FileTime`. W przykładzie ustawiliśmy ostatnią datę modyfikacji w godzinach od `epoch`. Oczywiście obie te metody wyrzucą `IOException` jeżeli plik nie zostanie znaleziony.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    FileTime lastModifiedTime = Files.getLastModifiedTime(path);
    System.out.println(lastModifiedTime);

    Path path1 = Files.setLastModifiedTime(path, FileTime.from(18284L, TimeUnit.HOURS));
    FileTime lastModifiedTime2 = Files.getLastModifiedTime(path1);
    System.out.println(lastModifiedTime2); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowane: `1972-02-01T20:00:00Z`

# isDirectory()

Metoda jak sama nazwa wskazuje, czy podana ścieżka odnosi się do katalogu.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka");

boolean directory1 = Files.isDirectory(path1);
System.out.println("directory1: " + directory1); ①
boolean directory2 = Files.isDirectory(path2);
System.out.println("directory2: " + directory2); ②
```

① Na ekranie zostanie wydrukowane: `directory1: false`

② Na ekranie zostanie wydrukowane: `directory2: true`

# isRegularFile()

Ta metoda z kolei wskazuje, czy podana ścieżka odnosi się do pliku.

```
Path path1 = Paths.get("zajavka/myFile.txt");
Path path2 = Paths.get("zajavka");

boolean regularFile1 = Files.isRegularFile(path1);
System.out.println("regularFile1: " + regularFile1); ①
boolean regularFile2 = Files.isRegularFile(path2);
System.out.println("regularFile2: " + regularFile2); ②
```

① Na ekranie zostanie wydrukowane: `regularFile1: true`

② Na ekranie zostanie wydrukowane: `regularFile2: false`

# isHidden()

Ta metoda pozwala sprawdzić, czy plik jest ukryty. Dlatego, że możemy ukrywać pliki na dysku ☺.

```
Path path1 = Paths.get("zajavka/myFile.txt");

try {
    boolean hidden1 = Files.isHidden(path1);
    System.out.println("hidden1: " + hidden1); ①
} catch (IOException e) {
    e.printStackTrace();
}
```

① Na ekranie zostanie wydrukowane: `hidden1: false`

# isReadable() i isExecutable()

W niektórych systemach operacyjnych użytkownik może nie mieć uprawnień aby odczytać plik, a tym bardziej aby go uruchomić.

W przypadku `isExecutable()` nie jest natomiast sprawdzane czy rozszerzenie pliku jest 'uruchamialne'. Czyli, przykładowo rozszerzenie `.png` nie uruchamia nam żadnego programu ale `.exe` już tak. Dlatego w poniższym przypadku mamy dwukrotnie `true`.

```
Path path1 = Paths.get("zajavka/myFile.txt");
boolean readable1 = Files.isReadable(path1);
System.out.println("readable1: " + readable1); ①

boolean executable1 = Files.isExecutable(path1);
System.out.println("executable1: " + executable1); ②
```

① Na ekranie zostanie wydrukowane: `readable1: true`

② Na ekranie zostanie wydrukowane: `executable1: true`

# Przykładowe metody wprowadzone w Java 8

## list()

Metoda, która służy do listowania zawartości podanej ścieżki. Przeszukiwane jest tylko jedno zagłębienie katalogu, to znaczy, że jeżeli natrafimy na inny katalog, to jego zawartość nie zostanie uwzględniona. Zwracany jest `Stream<Path>`, dlatego wspomniałem o Javie 8. Mając taki `Stream`, możemy wykonywać operacje takie jak `map()`, `filter()` itp. Warto też zauważyć, że metoda ta wyrzuca `IOException`, który musimy obsłużyć.

```
Path path = Paths.get("zajavka");
try {
    Stream<Path> list = Files.list(path);
    List<Path> absolutes = list
        .filter(p -> Files.isRegularFile(p))
        .map(p -> p.toAbsolutePath())
        .collect(Collectors.toList());
    absolutes.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Wynikiem działania powyższego kodu będzie wydrukowanie na ekranie wszystkich ścieżek absolutnych do plików, które znajdują się w katalogu `zajavka` w obecnym **WD**.

## lines()

Wcześniej wspominałem o metodzie `readAllLines()` i wspomniałem, że nie poleca się jej używać do dużych plików, ze względu na to, że cała zawartość pliku jest ładowana do pamięci jednocześnie. Istnieje natomiast taka metoda jak `Files.lines()`, która zwraca `Stream<String>` i obchodzi ten problem. Metoda ta ładuje do pamięci tylko fragment pliku, a nie jego całość.

```
Path path = Paths.get("zajavka/myFile.txt");
try {
    Stream<String> lines = Files.lines(path);
    List<String> mapped = lines
        .filter(line -> line.contains("2") || line.contains("4")) ①
        .map(String::toUpperCase) ②
        .collect(Collectors.toList());
    mapped.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

① Zostaw tylko te linijki, które zawierają w sobie wartość `2` lub `4`.

② Zamień wszystkie pozostałe linijki na pisane z wielką literą.

Wynikiem działania powyższego kodu będzie wydrukowanie na ekranie linijek z pliku `myFile.txt`, który znajdują się w katalogu `zajavka` w obecnym **WD**. Zostaną wydrukowane tylko te linijki, które zawierają w sobie wartość `2` lub `4`. Zostaną one wydrukowane w całości wielką literą.

# Porównanie starych i nowych metod

| Stara metoda | NIO.2 |
|---|---|
| file.getName() | path.getFileName() |
| file.getAbsolutePath() | path.toAbsolutePath() |
| file.exists() | Files.exists(path) |
| file.isDirectory() | Files.isDirectory(path) |
| file.isFile() | Files.isRegularFile(path) |
| file.isHidden() | Files.isHidden(path) |
| file.length() | Files.size(path) |
| file.lastModified() | Files.getLastModifiedTime(path) |
| file.setLastModified(time) | Files.setLastModifiedTime(path, time) |
| file.delete() | Files.delete(path) |
| file.renameTo(file) | Files.move(source, destination) |
| file.mkdir() | Files.createDirectory(path) |
| file.mkdirs() | Files.createDirectories(path) |
| file.listFiles() | Files.list(path) |