JAVA SE 8 PROGRAMMER II

ZAKRES SZKOLENIA – DZIEŃ 3

- API dot. Daty / czasu w Javie SE 8

  klasy: LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration; obsługa stref czasowych; tworzenie i praca z obiektami reprezentującymi odcinki czasu z wykorzystaniem klas: Period, Duration, TemporalUnit

- Podstawy Java IO

  odczyt i zapis danych z/do konsoli; użycie klas z pakietu java.io: BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream oraz PrintWriter

- Plikowe I/O w Javie (NIO.2)

  użycie interfejsu Path do operacji na ścieżkach do plików i folderów; użycie klasy Files do operacji takich jak odczyt, usunięcie, kopiowanie pliku bądź folderu;

  użycie Stream API z NIO.2

API DATY I CZASU W JAVIE 8

Jedną z nowości wprowadzonych w Javie 8 jest **nowe API związane z obsługą dat i czasu**, znane też jako **JSR-310**. Jest ono łatwe do zrozumienia, logiczne i w dużej mierze podobne do biblioteki Joda (dostępna jeszcze przed wejściem java 8). W trakcie szkolenia omówione zostaną główne klasy do obsługi daty/czasu począwszy od czasu lokalnego.

OBSŁUGA CZASU LOKALNEGO (BEZ STREFY CZASOWEJ)

Do głównych klas obsługujących **datę i czas lokalny** (bez stref czasowych) należą:

- java.time.**LocalTime**
- java.time.**LocalDate**
- java.time.**LocalDateTime**
- java.time.**Instant**

LOCALTIME

Klasa **LocalTime** reprezentuje czas, bez powiązania go z konkretną strefą czasową, czy nawet datą. Jest to czas bez uwzględnienia strefy czasowej w systemie kalendarzowym **ISO-8601**. Posiada ona kilka bardzo użytecznych metod:

- **now()** - metoda statyczna, która zwraca aktualny czas. Domyślnym formatem jest **HH:mm:ss.mmm** (godziny:minuty:sekundy.milisekundy).

```
1  LocalTime localTime = LocalTime.now();
2  System.out.println("Current time: " + localTime);
3  // Current time: 22:34:27.106
```

LOCALTIME

- **withHour(), withMinute(), withSecond(), withNano()** - ustawiają wskazaną przez nas odpowiednio godzinę, minuty, sekundy i nanosekundy w aktualnym obiekcie LocalTime, np.:

```
1  LocalTime localTime = LocalTime.now()
2          .withSecond(0) // ustawiamy sekundy na wartość 0
3          .withNano(0);  // ustawiamy nanosekundy na wartość 0
4  System.out.println("Current time: " + localTime);
5  // Current time: 22:41
```

LOCALTIME

- **plusNanos(x), plusSeconds(x), plusMinutes(x), plusHours(x), minusNanos(x), minusSeconds(x), minusMinutes(x), minusHours(x)** – dodawanie (odejmowanie) nanosekund, sekund, minut, godzin do (od) zadanego czasu, np.:

```
1  LocalTime now = LocalTime.now();
2  System.out.println("Current time: " + now);
3  // Current time: 22:49:01.241
4  now = now.plusMinutes(10).plusHours(1);
5  System.out.println("Current time after addition: " + now);
6  // Current time after addition: 23:59:01.241
```

LOCALTIME

- **getHour()** - zwraca godzinę czasu, którą reprezentuje obiekt
- **getMinute()** - zwraca minutę czasu, którą reprezentuje obiekt
- **getSecond()** - zwraca sekundę czasu, którą reprezentuje obiekt

```
1  LocalTime now = LocalTime.now();
2  String formattedTime = now.getHour() + ":" + now.getMinute() + ":" + now.getSecond();
3  System.out.println(formattedTime);  // 22:55:26
```

LOCALDATE

Istotą istnienia klasy **LocalDate** jest **reprezentacja daty** (rok, miesiąc, dzień). Obiekt ten **nie uwzględnia i nie przechowuje czasu** (np. aktualnej godziny) **ani strefy czasowej**. Data domyślnie przechowywana jest w formacie **ISO-8601**. Poniżej przedstawione są główne metody operujące na obiektach lokalnej daty:

- **now()** - metoda statyczna, która zwraca bieżącą datę. **Domyślnym formatem jest YYYY-mm-dd**, np.:

```
1  LocalDate now = LocalDate.now();
2  System.out.println(now);
3  // 2020-03-27
```

LOCALDATE

- **of(year, month, dayOfMonth)** - tworzy obiekt reprezentujący datę (rok, miesiąc, dzień). Miesiąc można przedstawić za pomocą **enuma** **java.time.Month** lub indeksu miesiąca, np.:

```
1  LocalDate localDate = LocalDate.of(2020, Month.MARCH, 28);
2  System.out.println(localDate);
3  // 2020-03-28
```

- **java.time.Month** jest enumeracją, w związku z tym:

```
12:   Month month = Month.JANUARY;
13:   boolean b1 = month == 1;             // DOES NOT COMPILE
14:   boolean b2 = month == Month.APRIL;   // false
```

LOCALDATE

- **getYear()** - zwraca int reprezentujący rok
- **getMonth()** - zwraca miesiąc za pomocą obiektu java.time.Month
- **getDayOfYear()** - zwraca int informujący, który to dzień roku
- **getDayOfMonth()** - zwraca int reprezentujący dzień miesiąca
- **getDayOfWeek()** - zwraca dzień tygodnia wykorzystując enum java.time.DayOfWeek

LOCALDATETIME

Istotą istnienia klasy **LocalDateTime** jest reprezentacja daty (rok, miesiąc, dzień) **oraz czasu** (godzina, minuta, sekunda, milisekunda). Jest to format bez uwzględnienia strefy czasowej w systemie kalendarzowym **ISO-8601**.
Oto wybrane metody z tej klasy, które są najczęściej używane:

- **now()** - metoda statyczna, która zwraca aktualną datę i czas. Domyślnym formatem jest **YYYY-MM-ddThh:mm:ss.mmm**, np.

```
1  LocalDateTime localDateTime = LocalDateTime.now();
2  System.out.println(localDateTime);
3  // 2020-03-28T20:25:16.124
```

LOCALDATETIME

- **of(year, month, dayOfMonth, hour, minutes, seconds, milliseconds)** - statyczna metoda zwracająca lokalną datę i czas według zadanych parametrów (rok, miesiąc, dzień miesiąca, godzina, minuty, sekundy, milisekundy). Istnieją również przeciążone odpowiedniki tej metody ze zmienną liczbą parametrów. W drukowanej wartości zauważmy znak **T** - jest to **umowny separator oddzielający wartość daty od czasu**.

```
1 LocalDateTime localDateTime = LocalDateTime.of(2020, Month.MARCH, 28, 20, 0, 10, 0);
2 System.out.println(localDateTime);
3 // 2020-03-28T20:00:10
```

LOCALDATETIME

```
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime time)
```

MODYFIKACJA DATY I CZASU

- Wszyskie klasy odnoszące się do daty i czasu są klasami **immutable**. Oznacza to, że wyniki wywołania poszczególnych metod **należy przypisać do zmiennych**!

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date);          // 2014-01-20
14: date = date.plusDays(2);
15: System.out.println(date);          // 2014-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date);          // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date);          // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date);          // 2019-02-28
```

MODYFIKACJA DATY I CZASU

	Can Call on `LocalDate`?	Can Call on `LocalTime`?	Can Call on `LocalDateTime` or `ZonedDateTime`?
`plusYears/ minusYears`	Yes	No	Yes
`plusMonths/ minusMonths`	Yes	No	Yes
`plusWeeks/ minusWeeks`	Yes	No	Yes
`plusDays/ minusDays`	Yes	No	Yes
`plusHours/ minusHours`	No	Yes	Yes
`plusMinutes/ minusMinutes`	No	Yes	Yes
`plusSeconds/ minusSeconds`	No	Yes	Yes
`plusNanos/ minusNanos`	No	Yes	Yes

KLASY ZWIĄZANE ZE STREFAMI CZASOWYMI

- **ZonedDateTime** — data i czas powiązane z konkretną strefą czasową, rozumianą jako przybliżoną lokalizację geograficzną (np. Europa/Warszawa). Takie powiązanie pozwala także na uwzględnianie kwestii takich jak czas letni/zimowy itp.
- **ZoneId** — identyfikator strefy czasowej jako rejonu geograficznego (np. ZoneId.of(“Europe/Paris”) zwróci identyfikator odpowiadający strefie czasowej obowiązującej w Paryżu)

KLASY ZWIĄZANE ZE STREFAMI CZASOWYMI

```java
// aktualna z domyślną strefą
ZonedDateTime zonedDateTime1 = ZonedDateTime.now();
System.out.println("Aktualny czas w strefie komputera (serwera): " + zonedDateTime1);

// aktualna w konretnej strefie
ZonedDateTime zonedDateTime2 = ZonedDateTime.now(ZoneId.of("Poland"));
System.out.println("Aktualny czas w strefie czasowej Polski: " + zonedDateTime2);

// na podstawie LocalDateTime w konkretnej strefie
LocalDateTime localDateTime = LocalDateTime.parse("2015-05-30T14:20:50");
ZonedDateTime zonedDateTime3 = ZonedDateTime.of(localDateTime, ZoneId.of("Japan"));
System.out.println("Data '2015-05-30T14:20:50' w strefie czasowej Japoni: " + zonedDateTime3);
```

MODYFIKACJE STREFY CZASOWEJ

Wyobraź sobie, że wylatujesz z Warszawy 22.05.2019 o godzinie 15:00, lecisz do Tokio przez 7 godzin. Która godzina i jaka data będzie na miejscu (w obecnej strefie czasowej czyli w Tokyo)?

```
1  LocalDateTime localDepartureTime = LocalDateTime.parse("2019-05-22T15:00:00");
2  ZoneId departureZone = ZoneId.of("Europe/Warsaw");
3  ZonedDateTime departureTime = ZonedDateTime.of(localDepartureTime,departureZone);
4
5  System.out.println("Data i czas odlotu w strefie czasowej Polski: " + departureTime);
6
7  ZonedDateTime arrivalTime = departureTime.plusHours(7);
8  System.out.println("Data i czas przylotu w strefie czasowej Polski: " + arrivalTime);
9
10 ZonedDateTime arrivalTimeTokyo = arrivalTime.withZoneSameInstant(ZoneId.of("Asia/Tokyo"));
11 System.out.println("Data i czas przylotu w strefie czasowej Tokyo: " + arrivalTimeTokyo
```

KLASY REPREZENTUJĄCE ODSTĘPY CZASOWE

JSR-310 (JSR –Java Specification Request) wprowadza pojęcie odstępu czasu jako czasu, który upłynął pomiędzy momentami A i B. Służą do tego dwie klasy:

- **java.time.Duration**
- **java.time.Period**

Różnią się jedynie jednostkami (pozwalają reprezentować odpowiednio: jednostki **czasu** (**Duration**) i np. **miesiące czy lata** (**Period**)).

KLASY REPREZENTUJĄCE ODSTĘPY CZASOWE

Rozróżnienie na Duration i Period wynika z prostego faktu — wszystkie długości wyrażane poprzez **Duration**, mają **swoją reprezentację w podstawowych jednostkach czasu** (czyli np. w sekundach), podczas gdy te wyrażane przez **Period** (miesiąc, rok, wiek, milenium) **mogą mieć różną długość realną** (przez różną liczbę dni w miesiącach, lata przestępne itp.).

```
1  System.out.println(Duration.ofHours(10).toMinutes());
2  // 10 godzin wyrażona w minutach: 600
3
4  // W przykładzie poniżej została wyliczona różnica czasu
5  // w minutach pomiędzy czasem obecnym a czasem o 2 dni późniejszym
6  System.out.println(Duration.between(
7      LocalDateTime.now(), LocalDateTime.now().plusDays(2)).toMinutes()); // 2880
8
9  // Poniżej została wyliczona liczba miesięcy pomiędzy dwoma datami
10 System.out.println(Period.between(LocalDate.now(), LocalDate.now().plusDays(100))
11     .getMonths()); // 3
```

PERIOD

Następujące metody umożliwiają nam kreację obiektu **Period**:

```
Period annually = Period.ofYears(1);              // every 1 year
Period quarterly = Period.ofMonths(3);            // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3);       // every 3 weeks
Period everyOtherDay = Period.ofDays(2);          // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7);    // every year and 7 days
```

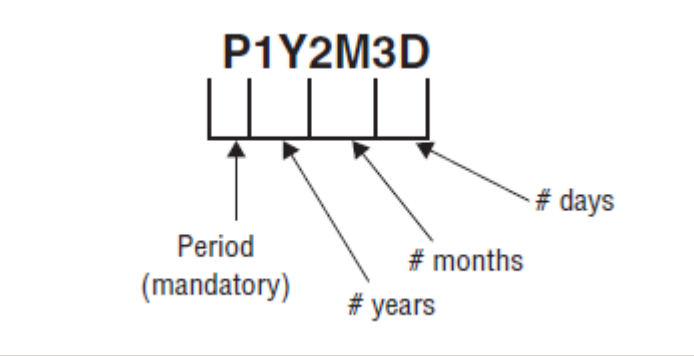
Uwaga. Nie możemy posługiwać się wywołaniem łańcuchowym.

```
Period wrong = Period.ofYears(1).ofWeeks(1);          // every week
```

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(1);
```

PERIOD - FORMAT

```
1  Period period = Period.of(1, 2, 3);
2  System.out.println(period);
```



DURATION

Następujące metody umożliwiają nam kreację obiektu **Duration**:

```
Duration daily = Duration.ofDays(1);                // PT24H
Duration hourly = Duration.ofHours(1);              // PT1H
Duration everyMinute = Duration.ofMinutes(1);       // PT1M
Duration everyTenSeconds = Duration.ofSeconds(10);  // PT10S
Duration everyMilli = Duration.ofMillis(1);         // PT0.001S
Duration everyNano = Duration.ofNanos(1);           // PT0.000000001S
```

DURATION

Klasa Duration dostarcza ponadto generyczną metodę fabrykującą umożliwiającą kreację obiektu.

```
Duration daily = Duration.of(1, ChronoUnit.DAYS);
Duration hourly = Duration.of(1, ChronoUnit.HOURS);
Duration everyMinute = Duration.of(1, ChronoUnit.MINUTES);
Duration everyTenSeconds = Duration.of(10, ChronoUnit.SECONDS);
Duration everyMilli = Duration.of(1, ChronoUnit.MILLIS);
Duration everyNano = Duration.of(1, ChronoUnit.NANOS);
```

Przy pomocy klasy ChronoUnit możemy wyznaczać „odległość” pomiędzy obiektami czasowymi:

```
LocalTime one = LocalTime.of(5, 15);
LocalTime two = LocalTime.of(6, 30);
LocalDate date = LocalDate.of(2016, 1, 20);

System.out.println(ChronoUnit.HOURS.between(one, two));   // 1
System.out.println(ChronoUnit.MINUTES.between(one, two)); // 75
System.out.println(ChronoUnit.MINUTES.between(one, date)); // DateTimeException
```

DURATION / PERIOD

	Can Use with Period?	Can Use with Duration?
`LocalDate`	Yes	No
`LocalDateTime`	Yes	Yes
`LocalTime`	No	Yes
`ZonedDateTime`	Yes	Yes

CHRONOUNIT

Na potrzeby **reprezentacji różnych jednostek czasowych** powstała klasa ChronoUnit ze stałymi reprezentującymi konkretne jednostki (np. **ChronoUnit.YEARS**).

```
// dodaj konkretną liczbę danych jednostek czasowych
LocalDate plus(long amountToAdd, TemporalUnit unit);
// dodaj konkretną liczbę danych jednostek czasowych
LocalDate minus(long amountToSubtract, TemporalUnit unit);
```

```
LocalDate localDate = LocalDate.now();
System.out.println("Aktualna data: " + localDate);

LocalDate date6YearsAfter = localDate.plus(6, ChronoUnit.YEARS);
System.out.println("Data za 6 lat: " + date6YearsAfter);

LocalDate date50DaysBefore = localDate.minus(50, ChronoUnit.DAYS);
System.out.println("Data sprzed 50 dni: " + date50DaysBefore);
```

INSTANT

Klasa ta wyróżnia się tym, że reprezentuje konkretny i **jednoznacznie określony punkt w czasie** (**z dokładnością do nanosekundy**). Drugą ważną cechą jest to, że nie jest ona powiązana z konceptem dni czy lat, a jedynie z uniwersalnym czasem, tzw. UTC . W skrócie przechowuje ona **wewnętrznie liczbę sekund** (z dokładnością do nanosekund) od pewnego ustalonego punktu w czasie (1 stycznia 1970 roku — tzw. Epoch time). Najlepiej nadaje się do reprezentowania czasu w sposób, który będzie przetwarzany przez system i nie będzie wyświetlany użytkownikom końcowym.

INSTANT

Obiekt Instant reprezentuje określony punkt czasu w strefie GMT. Przykład:

```
Instant now = Instant.now();
// do something time consuming
Instant later = Instant.now();

Duration duration = Duration.between(now, later);
System.out.println(duration.toMillis());
```

Uwaga:

Nie można przeprowadzić prostej konwersji pomiędzy obiektem LocalDateTime oraz Instant. LocalDateTime nie posiada informacji o strefie czasowej (**nie jest tym samym interpretowany jak ten sam moment czasu**).

INSTANT

```
1  LocalDateTime localDateTime = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
2  System.out.println(localDateTime); // 2020-04-19T18:33:29.116691800
3
4  LocalTime localTime = LocalTime.ofInstant(instant, ZoneId.of("CET"));
5  System.out.println(localTime); // 18:33:29.116691800
6
7  LocalDate localDate = LocalDate.ofInstant(
8      instant, ZoneId.ofOffset("UTC", ZoneOffset.ofHours(2)));
9  System.out.println(localDate); // 2020-04-19
```

METODY PODSUMOWUJĄCE

Metoda	Instant	Local-Time	Local-DateTime	Zoned-DateTime	Typ zwracany	Opis
(statycz-na) now()	✔	✔	✔	✔	(różne)	Zwraca instancje danego typu opisującą 'ter-az'
toInstant()	✖	✖	✔	✔	Instant	Zwraca obiekt typu Instant reprezentujący ten sam moment
toLocalDate-Time()	✖	✖	✖	✔	LocalDate-Time	Zwraca obiekt reprezentujący lokalne datę i czas w tym samym momencie
toLocal-Time()	✖	✖	✔	✔	LocalTime	Zwraca obiekt reprezentujący lokalny czas w tym samym momencie
toLocal-Date()	✖	✖	✔	✔	LocalDate	Zwraca obiekt reprezentujący lokalną datę w tym samym momencie
atZone()	✔	✖	✔	✖	ZonedDate-Time	Zwraca obiekt reprezentujący określony moment lub lokalny czas jako ZonedDateTime

FORMAT WYŚWIETLANEJ DATY

Do formatowania obiektów typu: **LocalDate, LocalTime, LocalDateTime** służy metoda **format(formatter)**.

Przykład dla lokalnego czasu znajduje się poniżej:

```
1 LocalTime localTime= LocalTime.now();
2 String formattedLocalTime = localTime.format(DateTimeFormatter.ISO_LOCAL_TIME);
3 System.out.println(formattedLocalTime); // 21:11:00.024
```

Poza gotowymi formatami, możemy również przygotować własne implementacje. W tym celu musimy wykorzystać specjalne symbole, które mają określone znaczenie i reprezentację, np.

```
1 String date = LocalDate.now().format(DateTimeFormatter.ofPattern("MM:YYYY:dd"));
2 System.out.println(date); // 04:2020:19
```

FORMAT WYŚWIETLANEJ DATY

Listę i opis wszystkich dostępnych symboli służących do formatowania obiektów LocalDate LocalTime LocalDateTime znajdziemy tutaj: **https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html**

Java 8 wprowadziła klasę **DateTimeFormatter**, która jest odpowiednikiem klasy **SimpleDateFormat**. Różnica polega na tym, że nowy "formater" potrafi współpracować z nowymi typami czasowymi (**ZonedDateTime, LocalDateTime**).

FORMAT WYŚWIETLANEJ DATY

```java
// formatowanie daty
DateTimeFormatter dtf1 = DateTimeFormatter.ofPattern("dd.MM.yyyy hh:mm");
ZonedDateTime zonedDateTime = ZonedDateTime.now();
System.out.println("Aktualna data (format domyślny): " + zonedDateTime);
System.out.println("Aktualna data (format dd.MM.yyyy hh:mm): " + dtf1.format(zonedDateTime));

// parsowanie daty - gotowy formatter (yyyy-MM-ddThh:mm)
DateTimeFormatter dtf2 = DateTimeFormatter.ISO_DATE_TIME;
LocalDateTime localDateTimeParsed = LocalDateTime.parse("2019-10-25T18:20", dtf2);
System.out.println("Sparsowana LocalDateTime: " + localDateTimeParsed);

// parsowanie daty - własny formatter z konkretną strefą czasową
DateTimeFormatter dtf3 = DateTimeFormatter.ofPattern("MM/dd/yyyy HH:mm")
    .withZone(ZoneId.of("Europe/Warsaw"));
ZonedDateTime zonedDateTimeParsed = ZonedDateTime.parse("10/25/2019 18:20", dtf3);
System.out.println("Sparsowana ZonedDateTime: " + zonedDateTimeParsed);
```

PODSTAWY JAVA IO

Java udostępnia wiele mechanizmów do obsługi operacji wejścia/wyjścia. Są to, m.in. **strumienie oraz zaawansowane mechanizmy serializacji**:

- **strumieniowość** jest realizowana przez klasy **Java IO**, znajdujące się w pakiecie **java.io**,
- mechanizmy wykorzystujące **buforowanie**, realizowane są przez komponenty **Java NIO**, znajdujące się w pakiecie **java.nio**.

KLASA FILE

java.io.File:

- reprezentuje ścieżkę do pliku lub katalogu w systemie plików
- **bezpośrednio nie służy** do odczytywania zawartości pliku
- umożliwia odczytywanie informacji na temat istniejących plików lub katalogów
- umożliwia **odczytywanie zawartości** katalogów
- umożliwia tworzenie oraz usuwanie plików i katalogów

KREACJA OBIEKTU KLASY FILE

- Kreacja w oparciu o łańcuch znaków reprezentujący bezwzględną lub względną ścieżkę do pliku
- Separator plików (zależny od systemu operacyjnego) uzyskujemy poprzez:

```
1  System.getProperty("file.separator")
```

- Weryfikację istnienia pliku realizujemy np.:

```
1  File file = new File("/home/zoo.txt");
2  System.out.println(file.exists());
```

- Pozostałe konstruktory tworzące obiekt klasy File:

```
1  File parent = new File("/home/test");
2  File child = new File(parent, "/data/zoo.txt");
```

KREACJA OBIEKTU KLASY FILE

- **Uwaga**. Łańcuch znaków przekazywany do konstruktora klasy File może reprezentować ścieżkę bezwzględną lub ścieżkę względną.
- **Uwaga**. W sytuacji, w której obiekt „parent” będzie wartością nieokreśloną (null), wówczas zostanie pominięty.

WYBRANE METOD KLASY FILE

`exists()`	Returns `true` if the file or directory exists.
`getName()`	Returns the name of the file or directory denoted by this path.
`getAbsolutePath()`	Returns the absolute pathname string of this path.
`isDirectory()`	Returns `true` if the file denoted by this path is a directory.
`isFile()`	Returns `true` if the file denoted by this path is a file.
`length()`	Returns the number of bytes in the file. For performance reasons, the file system may allocate more bytes on disk than the file actually uses.
`lastModified()`	Returns the number of milliseconds since the epoch when the file was last modified.
`delete()`	Deletes the file or directory. If this pathname denotes a directory, then the directory must be empty in order to be deleted.
`renameTo(File)`	Renames the file denoted by this path.
`mkdir()`	Creates the directory named by this path.
`mkdirs()`	Creates the directory named by this path including any nonexistent parent directories.
`getParent()`	Returns the abstract pathname of this abstract pathname's parent or `null` if this pathname does not name a parent directory.
`listFiles()`	Returns a `File[]` array denoting the files in the directory.

WYBRANE METODY KLASY FILE - PRZYKŁAD

```java
File file = new File("C:\\data\\zoo.txt");
System.out.println("File exists: " + file.exists());;
if (file.exists()) {
    System.out.println("Absolute path: " + file.getAbsolutePath());
    System.out.println("Is directory: " + file.isDirectory());
    System.out.println("Parent path: " + file.getParent());

    if (file.isFile()) {
        System.out.println("File size: " + file.length());
        System.out.println("File last modified at: " + file.lastModified());
    } else {
        for (File subFile : file.listFiles()) {
            System.out.println("\t" + subFile.getName());
        }
    }
}
```

PRZETWARZANIE STRUMIENIOWE

Java IO jest zorientowana na przetwarzanie strumieniowe, które polega na tym, że:

- **odczytywana jest pewna liczba bajtów** (jeden lub więcej)
- programista, wykorzystując dane ze strumienia, **nie ma możliwości się po nim „poruszać”**

Java NIO jest zorientowana na buforowanie, tzn. że:

- dane są najpierw **wczytywane do bufora** o pewnym rozmiarze
- następnie są przekazywane do dyspozycji programisty, który **może „poruszać” się po takim buforze**

BLOKOWANIE WĄTKU

Wykorzystanie strumieni Java IO powoduje zablokowanie aktualnie działającego wątku, tzn. korzystając z metod **read()** czy **write()** z klas z pakietu **java.io**, blokujemy dalsze wykonywanie się programu.

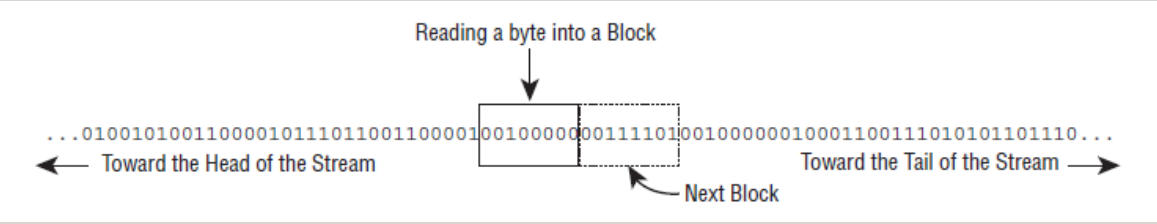
Tryb **nieblokujący Java NIO**, umożliwia wątkowi zażądanie odczytu danych z kanału i uzyskanie tylko tego, co jest obecnie dostępne lub wcale, jeśli żadne dane nie są obecnie dostępne. Zamiast pozostać zablokowanym, dopóki dane nie będą dostępne do odczytu, wątek może kontynuować pracę z czymś innym.

PRZETWARZANIE STRUMIENIOWE

Sposób przetwarzania strumienia danych jest różny w zależności od wykorzystywanej implementacji.

Możemy wyróżnić strategie:

- Odczytywanie kolejno **pojedynczych bajtów**
- Odczytywanie kolejno **pojedynczych znaków lub łańcuchów znaków**
- **Odczytywanie zestawu bajtów lub znaków** (chrakterystyczne dla strumieni „buffered”)



STRUMIEŃ BAJTÓW VS STRUMIEŃ ZNAKÓW

- Klasy bazujące wprost na strumieniach są wykorzystywane do **odczytu i zapisu binarnych typów danych** (InputStream, OutputStream)
- Klasy **Reader** oraz **Writer** są wykorzystywane do **zapisu i odczytu wyłącznie pojedynczych znaków i łańcuchów znaków** (Reader, Writer)

STRUMIENIE NISKO ORAZ WYSOKOPOZIOMOWE

- **Strumień niskopoziomowy** – bezpośrednia interakcja ze źródłem danych (plik, tablica, łańcuch znaków). Przykład: **FileInputStream** – bezpośrednio odczyt poszczególnych bajtów pliku
- **Strumień wysokopoziomowy** – zbudowany w oparciu o inny strumień przekazany jako parametr wywołania konstruktora („*wrapping*"). Poszczególne operacje są filtrowane i przekazywane docelowo do źródłowego strumienia. Przykład: **FileReader**oraz **BufferedReader**

```
try (
   BufferedReader bufferedReader = new BufferedReader(
                                       new FileReader("zoo-data.txt"))) {
   System.out.println(bufferedReader.readLine());
}
```

STRUMIENIE NISKO ORAZ WYSOKOPOZIOMOWE

- Najczęściej, „opakowywanie” strumienie niskopoziomowych strumieniami wysokopoziomowymi wiąże się z możliwością korzystania z rozszerzonego API. Przykład: metoda **readLine()** udostępniona w klasie **BufferedReader**
- Strumienie wysokopoziomowe mogą w ramach wywołania konstruktora przyjmować inne, wysokopoziomowe strumienie.

```
try (ObjectInputStream objectStream = new ObjectInputStream(
                                      new BufferedInputStream(
                                       new FileInputStream("zoo-data.txt")))) {
   System.out.println(objectStream.readObject());
}
```

BAZOWE KLASY STRUMIENI

- Cztery podstawowe klasy:
  InputStream, **OutputStream**, **Reader**, **Writer**
- Strategia nazewnicza klas potomnych:
  zachowanie **nazwy klasy abstrakcyjnej jako przyrostek (suffix) nazwy klasy potomnej**
  Przykłady: **ObjectInputStream**, **BufferedWriter**
  Wyjątek: **PrintStream**

```
new BufferedInputStream(new FileReader("zoo-data.txt"));     // DOES NOT COMPILE
new BufferedWriter(new FileOutputStream("zoo-data.txt"));    // DOES NOT COMPILE
new ObjectInputStream(new FileOutputStream("zoo-data.txt")); // DOES NOT COMPILE
new BufferedInputStream(new InputStream());                  // DOES NOT COMPILE
```

WŁAŚCIWOŚCI KLAS PAKIETU JAVA.IO

- ***InputStream** oraz ***OutputStream** – odczyt i zapis danych binarnych
- ***Reader** oraz ***Writer** – odczyt i zapis znaków lub łańcuchów znaków
- „Niskopoziomowe" strumienie bezpośrednio pracują na źródle danych
- „Wysokopoziomowe" strumienie są „zbudowane" na innych strumieniach (pełnią rolę tzw. „wrapper'ów")
- Obiekty klas **Buffered*** odczytują lub zapisują dane w grupach (bajtów lub znaków), co znacząco poprawia wydajność w przypadku sekwencyjnych systemów plików

`InputStream`	N/A	The abstract class all `InputStream` classes inherit from
`OutputStream`	N/A	The abstract class all `OutputStream` classes inherit from
`Reader`	N/A	The abstract class all `Reader` classes inherit from
`Writer`	N/A	The abstract class all `Writer` classes inherit from
`FileInputStream`	Low	Reads file data as bytes
`FileOutputStream`	Low	Writes file data as bytes
`FileReader`	Low	Reads file data as characters
`FileWriter`	Low	Writes file data as characters
`BufferedReader`	High	Reads character data from an existing `Reader` in a buffered manner, which improves efficiency and performance
`BufferedWriter`	High	Writes character data to an existing `Writer` in a buffered manner, which improves efficiency and performance
`ObjectInputStream`	High	Deserializes primitive Java data types and graphs of Java objects from an existing `InputStream`
`ObjectOutputStream`	High	Serializes primitive Java data types and graphs of Java objects to an existing `OutputStream`
`InputStreamReader`	High	Reads character data from an existing `InputStream`
`OutputStreamWriter`	High	Writes character data to an existing `OutputStream`
`PrintStream`	High	Writes formatted representations of Java objects to a binary stream
`PrintWriter`	High	Writes formatted representations of Java objects to a text-based output stream

FILEINPUTSTREAM FILEOUTPUTSTREAM

Klasy oparte o **InputStream** i **OutputStream** używają **niebuforowalnych** operacji wejścia/wyjścia. Oznacza to, że każde żądanie odczytu lub zapisu **jest obsługiwane bezpośrednio przez podstawowy system operacyjny**. Może to sprawić, że program będzie znacznie **mniej wydajny**, ponieważ każde takie żądanie często zezwala na dostęp do dysku, aktywność sieciową lub inną kosztowną operację.

```
import java.io.*;

public class CopyFileSample {
   public static void copy(File source, File destination) throws IOException {
      try (InputStream in = new FileInputStream(source);
         OutputStream out = new FileOutputStream(destination)) {
         int b;
         while((b = in.read()) != -1) {
            out.write(b);
         }
      }
   }

   public static void main(String[] args) throws IOException {
      File source = new File("Zoo.class");
      File destination = new File("ZooCopy.class");
      copy(source,destination);
   }
}
```

BUFFEREDINPUTSTREAM BUFFEREDOUTPUTSTREAM

W celu redukcji tego rodzaju obciążeń, platforma Java implementuje **buforowane strumienie wejścia/wyjścia**. Buforowane strumienie wejściowe odczytują dane z obszaru pamięci znanego jako **bufor**.

Program może przekonwertować strumień niebuforowany na buforowany za pomocą klas takich jak: **BufferedInputStream** i **BufferedOutputStream**

```
import java.io.*;

public class CopyBufferFileSample {
   public static void copy(File source, File destination) throws IOException {
      try (
         InputStream in = new BufferedInputStream(new FileInputStream(source));
         OutputStream out = new BufferedOutputStream(
                                                new FileOutputStream(destination))) {
         byte[] buffer = new byte[1024];
         int lengthRead;
         while ((lengthRead = in.read(buffer)) > 0) {
            out.write(buffer,0,lengthRead);
            out.flush();
         }
      }
   }
}
```

BUFFEREDREADER BUFFEREDWRITER

Odpowiednią konwersję strumieni niebuforowanych na strumienie buforowane jesteśmy w stanie również zrealizować w przypadku klas **Reader** oraz **Writer**. Posługujemy się tym razem klasami: **BufferedReader** oraz **BufferedWriter**.

```
import java.io.*;
import java.util.*;

public class CopyTextFileSample {
   public static List<String> readFile(File source) throws IOException {
      List<String> data = new ArrayList<String>();
      try (BufferedReader reader = new BufferedReader(new FileReader(source))) {
         String s;
         while((s = reader.readLine()) != null) {
            data.add(s);
         }
      }
      return data;
   }

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

   public static void main(String[] args) throws IOException {
      File source = new File("Zoo.csv");
      File destination = new File("ZooCopy.csv");
      List<String> data = readFile(source);
      for(String record: data) {
         System.out.println(record);
      }
      writeFile(data,destination);
   }
}
```

OPERACJE NA STRUMIENIACH - CLOSE

Wszystkie strumienie są **zasobami**.

- Wyciek zasobów w przypadku braku zamknięcia strumienia!
- Brak zamknięcia strumienia – możliwy lock na pliku założony przez systemy operacyjny (potencjalne zablokowanie pozostałych procesów)
- Interfejs **java.io.Closeable** – metoda **close()**
- Konstrukcja: **try-catch-finally**
- Konstrukcja: **try-with-resources**

OPERACJE NA STRUMIENIACH - FLUSH

- Wymuszenie zapisu danych strumienia
- Automatyczne wywołanie metody **flush** w sytuacji zamknięcia strumienia (**close**)
- Dobre praktyki oraz okoliczności wykorzystania metody **flush**

OPERACJE NA STRUMIENIACH - MARK

- **mark** - oznaczanie **pozycji wskaźnika** w strumieniu Klasy **InputStream** oraz **Reader** posiadają metody: **mark(int)** oraz **reset()**, które umożliwiają powrót do poprzedniej pozycji w strumieniu. Uwaga. Nie wszystkie klasy wspierają te operacje – **markSupported()**

```
InputStream is = ...
System.out.print ((char)is.read());
if(is.markSupported()) {
   is.mark(100);
   System.out.print((char)is.read());
   System.out.print((char)is.read());
   is.reset();
}
System.out.print((char)is.read());
System.out.print((char)is.read());
System.out.print((char)is.read());
```

OPERACJE NA STRUMIENIACH - SKIP

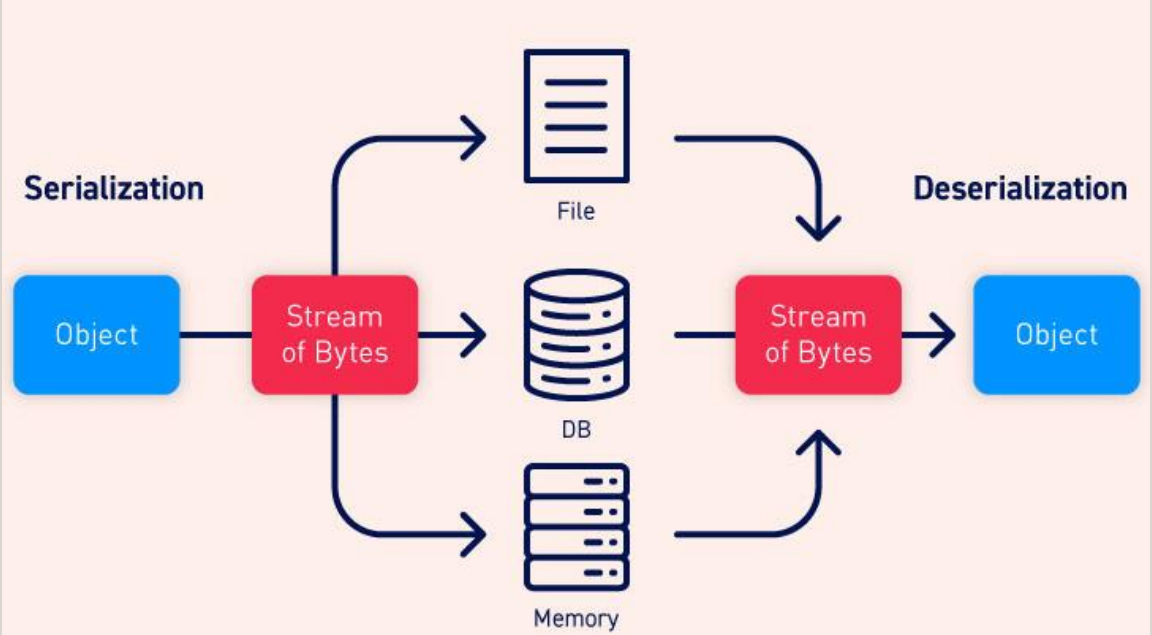
- **skip** – klasy **InputStream** oraz **Reader** umożliwiająca pominięcie podczas odczytu określonej liczby bajtów.

```
InputStream is = ...
System.out.print ((char)is.read());
is.skip(2)
is.read();
System.out.print((char)is.read());
System.out.print((char)is.read());
```

SERIALIZACJA ORAZ DESERIALIZACJA

- **Serializacja** jest mechanizmem, który realizuje konwersję **stanu obiektu na strumień bajtów**. **Deserializacja** jest natomiast procesem odwrotnym, który umożliwia **odtwarzanie stanu obiektu w pamięci na podstawie strumienia bajtów**.

- Powstały strumień bajtów jest **niezależny od platformy**, więc obiekt zserializowany na jednej platformie może zostać skutecznie zdeserializowany na innej platformie.



INTERFEJS SERIALIZABLE

- Interfejs **Serializable** włącza możliwość **serializacji** i **deserializacji** obiektów za pomocą, m.in. klasy **ObjectInputStream** i **ObjectOutputStream**, które zawierają **wysokopoziomowe API** do serializacji i deserializacji obiektów.

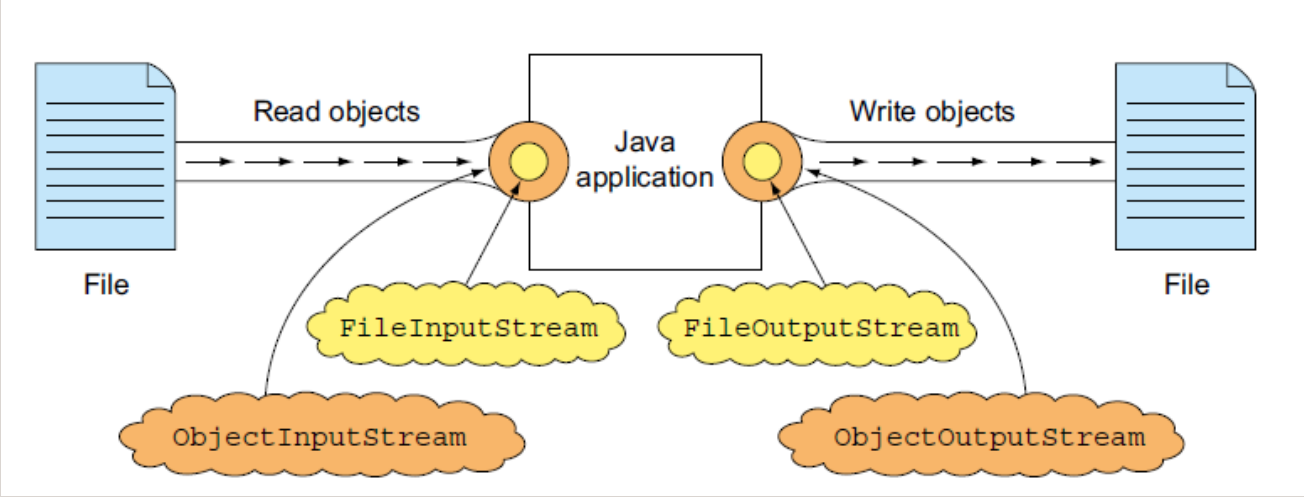
```
import java.io.Serializable;

public class Animal implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private int age;
   private char type;

   public Animal(String name, int age, char type) {
      this.name = name;
      this.age = age;
      this.type = type;
   }

   public String getName() { return name; }
   public int getAge() { return age; }
   public char getType() { return type; }

   public String toString() {
      return "Animal [name=" + name + ", age=" + age + ", type=" + type + "]";
   }
}
```

INTERFEJS SERIALIZABLE - UWAGI

- Serializable – „**marker interface**" (każda klasa może implementować dany interface – brak API)
- Wszystkie zmienne (instancyjne) w zakresie danej klasy **również powinny być serializowalne** – w innym przypadku otrzymujemy: **NotSerializableException**
- Duża część wbudowanych typów jest serializowalna (np. String).
- Zmienne **instancyjne transient** oraz **zmienne statyczne nie są serializowalne**
- Serializable – kiedy używać?
- Stała statyczna *serialVersionUUID* – kiedy używać?

SERIALIZACJA ORAZ DESERIALIZACJA OBIEKTÓW

- Pakiet java.io udostępnia dwie klasy do obsługi procesu serializacji oraz deserializacji obiektów: **ObjectInputStream** oraz **ObjectOutputStream.**




```
import java.io.*;
import java.util.*;
public class ObjectStreamSample {
   public static List<Animal> getAnimals(File dataFile) throws IOException,
                                                        ClassNotFoundException {
      List<Animal> animals = new ArrayList<Animal>();
      try (ObjectInputStream in = new ObjectInputStream(
            new BufferedInputStream(new FileInputStream(dataFile)))) {
         while(true) {
            Object object = in.readObject();
            if(object instanceof Animal)
               animals.add((Animal)object);
         }
      } catch (EOFException e) {
         // File end reached
      }
      return animals;
   }

   public static void createAnimalsFile(List<Animal> animals, File dataFile)
                                                              throws IOException {
      try (ObjectOutputStream out = new ObjectOutputStream(
            new BufferedOutputStream(new FileOutputStream(dataFile)))) {
         for(Animal animal: animals)
            out.writeObject(animal);
      }
   }

   public static void main(String[] args) throws IOException,
                                                        ClassNotFoundException {
      List<Animal> animals = new ArrayList<Animal>();
      animals.add(new Animal("Tommy Tiger",5,'T'));
      animals.add(new Animal("Peter Penguin",8,'P'));

      File dataFile = new File("animal.data");
      createAnimalsFile(animals,dataFile);
      System.out.println(getAnimals(dataFile));
   }
}
```

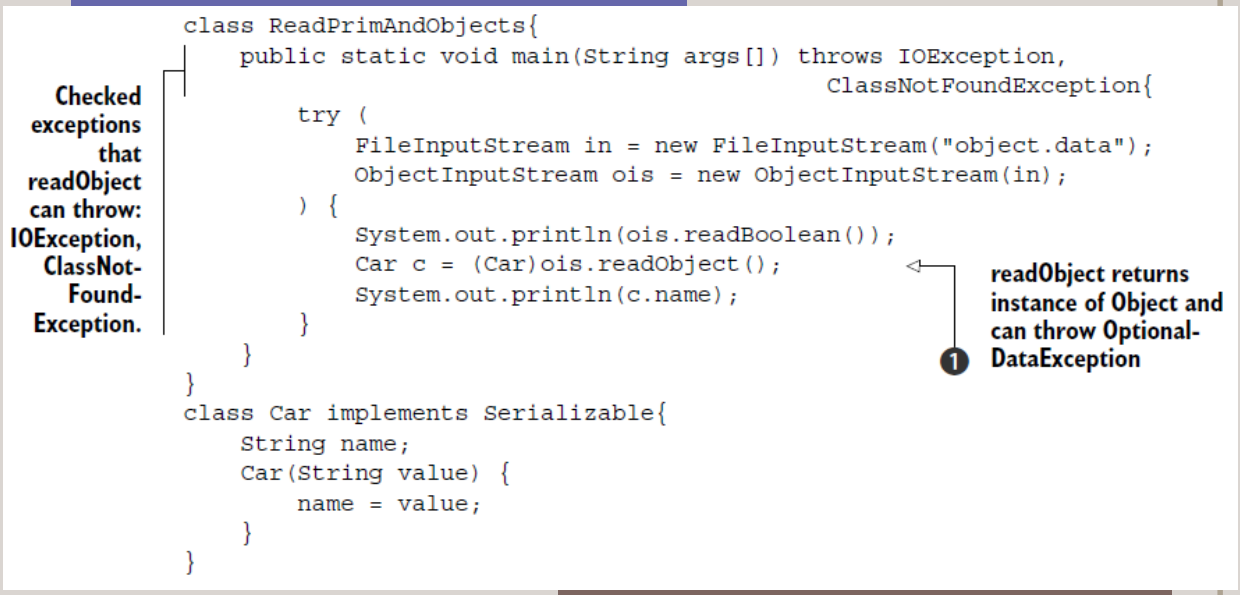
DESERIALIZACJA – KREACJA OBIEKTÓW

- **Konstruktor serializowanej klasy nie jest nigdy wywoływany**.
- Wywoływany jest pierwszy z napotkanych **konstruktorów bezparametrowych – klasy rodzica oznaczonej jako nieserializowana**
- **Statyczne zmienne oraz bloki inicjalizujące są ignorowane**

```
public class Animal implements Serializable {
   private static final long serialVersionUID = 2L;
   private transient String name;
   private transient int age = 10;
   private static char type = 'C';

   {this.age = 14;}

   public Animal() {
      this.name = "Unknown";
      this.age = 12;
      this.type = 'Q';
   }

   public Animal(String name, int age, char type) {
      this.name = name;
      this.age = age;
      this.type = type;
   }

   // Same methods as before
   ...
}
```

SERIALIZACJA ORAZ DESERIALIZACJA - UWAGI

- Klasy ObjectInputStream oraz ObjectOutputStream możemy wykorzytywac do odczytu zapisu obiektów oraz tzw. „prymitywów”.
- Wartości powinni być zawsze **odczytywane w tej samej kolejności, w której zostały zapisane.**




```
class ReadPrimAndObjects{
    public static void main(String args[]) throws IOException,
                                                   ClassNotFoundException{
        try (
            FileInputStream in = new FileInputStream("object.data");
            ObjectInputStream ois = new ObjectInputStream(in);
        ) {
            System.out.println(ois.readBoolean());
            Car c = (Car)ois.readObject();
            System.out.println(c.name);
        }
    }
}
class Car implements Serializable{
    String name;
    Car(String value) {
        name = value;
    }
}
```

DESERIALIZACJA – DOSTOSOWYWANIE PROCESU

```
public class Wheel {
  private int wheelSize;

  Wheel(int ws) { wheelSize = ws; }

  int getWheelSize() { return wheelSize; }

  public String toString() { return "wheel size: " + wheelSize; }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Unicycle implements Serializable {
  transient private Wheel wheel;

  Unicycle(Wheel wheel) { this.wheel = wheel; }

  public String toString() { return "Unicycle with " + wheel; }

  private void writeObject(ObjectOutputStream oos) {
    try {
      oos.defaultWriteObject();
      oos.writeInt(wheel.getWheelSize());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void readObject(ObjectInputStream ois) {
    try {
      ois.defaultReadObject();
      int wheelSize = ois.readInt();
      wheel = new Wheel(wheelSize);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

PRINTSTREAM ORAZ PRINWRITER

- Klasy **PrintStream** oraz **PrintWriter** reprezentują **wysokopoziomowe strumienie umożliwiające zapis sformatowanych danych** do bazowego strumienia tekstowego.
- **Metoda print**

```
PrintWriter out = new PrintWriter("zoo.log");

out.print(5); // PrintWriter method
out.write(String.valueOf(5)); // Writer method

out.print(2.0); // PrintWriter method
out.write(String.valueOf(2.0)); // Writer method

Animal animal = new Animal();
out.print(animal); // PrintWriter method
out.write(animal==null ? "null": animal.toString()); // Writer method
```

PRINTSTREAM ORAZ PRINWRITER

- Metoda **println** – metoda dodająca na końcu łańcucha znaków znak nowej linii (*System.getProperty(„line.separator”);*)
- Metody **format()** oraz **printf() –** metody identyczne w zachowaniu, parametrach wejściowych oraz parametrach wyjściowych

```
public PrintWriter format(String format, Object args...)
public PrintWriter printf(String format, Object args...)
```

```
import java.io.*;

public class PrintWriterSample {
   public static void main(String[] args) throws IOException {
      File source = new File("zoo.log");
      try (PrintWriter out = new PrintWriter(
            new BufferedWriter(new FileWriter(source)))) {
         out.print("Today's weather is: ");
         out.println("Sunny");
         out.print("Today's temperature at the zoo is: ");
         out.print(1/3.0);
         out.println('C');
         out.format("It has rained 10.12 inches this year");
         out.println();
         out.printf("It may rain 21.2 more inches this year");
      }
   }
}
```

KLASY PAKIETU JAVA.IO

ODCZYT DANYCH Z KONSOLI - BUFFEREDREADER

Zalety:

- **Dane wejściowe są buforowane** dla sprawnego odczytu

Wady:

- Kod jest zawiły i trudny do zapamiętania / utrzymania

```
1 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
2 String userInput = reader.readLine();
3 System.out.println("You entered the following: " + userInput);
```

ODCZYT DANYCH Z KONSOLI - SCANNER

Głównym przeznaczeniem klasy **Scanner** (dostępnej od wersji Java 1.5) jest **parsowanie typów prymitywnych** i łańcuchów przy użyciu wyrażeń regularnych, jednak może być ona również użyta do odczytywania danych wejściowych od użytkownika w linii poleceń.

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.println("Enter your nationality: ");
3 String nationality = scanner.nextLine();
4 System.out.println("Enter your age: ");
5 int age = scanner.nextInt();
6 System.out.println("Your nationality: " + nationality + ", age: " + age);
```

ODCZYT DANYCH Z KONSOLI - SCANNER

“Klasa Scanner odczytuje sformatowane dane wejściowe i konwertuje je do postaci binarnej. **Klasa może być używana do odczytywania danych wejściowych z konsoli, pliku, łańcuchów** [...]. Przykładowo możemy wykorzystać tę klasę do odczytywania liczb z klawiatury i przypisywania jej do odpowiedniej zmiennej”

H. Schildt, Java Kompendium Programisty

ODCZYT DANYCH Z KONSOLI - SCANNER

Zalety:

- Wygodne metody do parsowania prymitywów (**nextInt()**, **nextFloat()**, ...) z tokenizowanych danych wejściowych.
- Wyrażenia regularne mogą być użyte do wyszukiwania tokenów.

Wady:

- Metody odczytu **nie są zsynchronizowane**.

ODCZYT DANYCH Z KONSOLI - CONSOLE

Klasa **Console** została wprowadzona w **Javie 1.6**, i stała się **preferowanym sposobem na odczytywanie danych wejściowych użytkownika z linii poleceń**. Dodatkowo, może być używana do odczytywania danych wejściowych **typu hasło bez echa znaków** wprowadzonych przez użytkownika; składnia łańcucha formatu może być również używana (jak System.out.printf()).

```
1  Console console = System.console();
2  if (console != null) {
3      String userInput = console.readLine();
4      console.writer().println("You entered the following: " + userInput);
5      char[] password = console.readPassword("Enter password: ");
6      System.out.println("You entered password: " + String.valueOf(password));
7  } else {
8      System.out.println("No console!");
9  }
```

ODCZYT DANYCH Z KONSOLI - CONSOLE

Klasa **Console** została wprowadzona w **Javie 1.6**, i stała się **preferowanym sposobem na odczytywanie danych wejściowych użytkownika z linii poleceń**. Dodatkowo, może być używana do odczytywania danych wejściowych **typu hasło bez echa znaków** wprowadzonych przez użytkownika; składnia łańcucha formatu może być również używana (jak System.out.printf()).

```
1  Console console = System.console();
2  if (console != null) {
3      String userInput = console.readLine();
4      console.writer().println("You entered the following: " + userInput);
5      char[] password = console.readPassword("Enter password: ");
6      System.out.println("You entered password: " + String.valueOf(password));
7  } else {
8      System.out.println("No console!");
9  }
```

ODCZYT DANYCH Z KONSOLI - CONSOLE

Zalety:

- Odczytywanie hasła bez echa wprowadzonych znaków.
- Metody odczytu są zsynchronizowane.
- Możliwość użycia składni ciągu formatów.

Wady:

- **Nie działa w środowisku nieinteraktywnym (np. w IDE).**

CONSOLE – READER() ORAZ WRITER()

Klasa Console umożliwia dostęp do instancji **Reader** oraz **PrintWriter** odpowiednio w oparciu o metody **reader()** oraz **writer()**.

W celu wypisania danych na konsoli, możemy posłużyć się również metodami **format()** oraz **printf()**.

```
import java.io.*;

public class ConsoleSamplePrint {
   public static void main(String[] args) throws NumberFormatException,
IOException {
      Console console = System.console();
      if(console == null) {
         throw new RuntimeException("Console not available");
      } else {
         console.writer().println("Welcome to Our Zoo!");
         console.format("Our zoo has 391 animals and employs 25 people.");
         console.writer().println();
         console.printf("The zoo spans 128.91 acres.");
      }
   }
}
```

CONSOLE – ODCZYT HASŁA

```
import java.io.*;
import java.util.Arrays;

public class PasswordCompareSample {
   public static void main(String[] args) throws NumberFormatException,
                                                  IOException {
      Console console = System.console();
      if(console == null) {
         throw new RuntimeException("Console not available");
      } else {
         char[] password = console.readPassword("Enter your password: ");
         console.format("Enter your password again:  ");
         console.flush();
         char[] verify = console.readPassword();
         boolean match = Arrays.equals(password,verify);

         // Immediately clear passwords from memory
         for(int i=0; i<password.length; i++) {
            password[i]='x';
         }
         for(int i=0; i<verify.length; i++) {
            verify[i]='x';
         }

         console.format("Your password was "+(match ? "correct": "incorrect"));
      }
   }
}
```

PLIKOWE I/O W JAVIE (NIO.2)

Java NIO wprowadziła wiele uproszczonych mechanizmów do obsługi plików. Mechanizmy te zostały umieszczone w pakiecie java.nio.file, a punktem wejściowym do ich realizacji jest klasa Files.

PATH - WŁAŚCIWOŚCI

- **Path –** interfejs, reprezentuje ścieżkę do pliku lub katalogu
- Koncepcyjny **odpowiednik** klasy **java.io.File:**
  - w obydwu przypadkach możemy referować do pliku oraz katalogu
  - w obydwu przypadkach możemy referować do pliku / katalogu w oparciu o ścieżkę względną oraz bezwzględną
  - w obydwu przypadkach udostępniane są (w zdecydowanej większości) te same właściwości
- **Path** – wspiera linki symboliczne

PATH - WŁAŚCIWOŚCI

- **Path – nie jest plikiem**, a jedynie reprezentacją lokalizacji w zakresie całego systemu plików.
  Tym samym, większość operacji może zostać wykonana bez względu na to, czy referowany plik rzeczywiście istnieje.

PATH - KREACJA

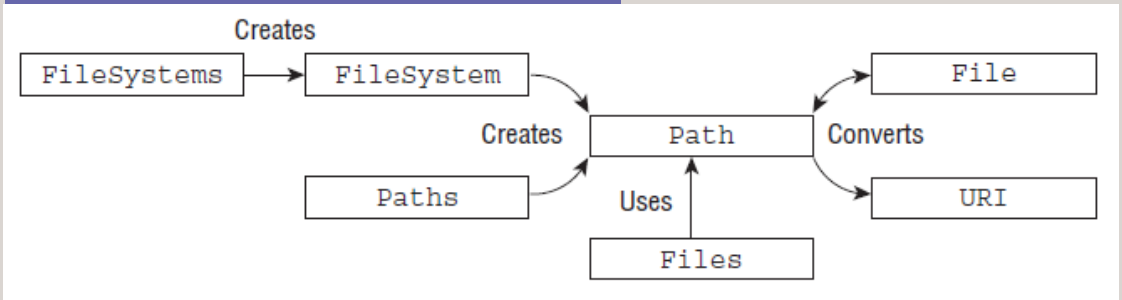
- Wykorzystanie klasy **Paths**

```
Path path1 = Paths.get("pandas/cuddly.png");

Path path2 = Paths.get("c:\\zooinfo\\November\\employees.txt");

Path path3 = Paths.get("/home/zoodirector");
```

```
Path path1 = Paths.get("pandas","cuddly.png");

Path path2 = Paths.get("c:","zooinfo","November","employees.txt");

Path path3 = Paths.get("/","home","zoodirector");
```

- **Uwaga:**

```
Paths path1 = Paths.get("/alligator/swim.txt");  // DOES NOT COMPILE

Path path2 = Path.get("/crocodile/food.csv");  // DOES NOT COMPILE
```



PATH - KREACJA

- Wykorzystanie klasy **Paths** z adresem URI
  - *Uniform Resource Identifier*

```
Path path1 = Paths.get(new URI("file://pandas/cuddly.png")); // THROWS EXCEPTION
                                                             // AT RUNTIME

Path path2 = Paths.get(new URI("file:///c:/zoo-info/November/employees.txt"));

Path path3 = Paths.get(new URI("file:///home/zoodirectory"));
```

- **Uwaga**: W przypadku adresu URI podajemy ścieżkę bezwzględną.
- Klasa Path posiada również metodę **toUri()** umożliwiającą konwersję odwrotną.

```
Path path4 = Paths.get(new URI("http://www.wiley.com"));
URI uri4 = path4.toUri();
```

PATH - KREACJA

- Obiekt implementujący interfejs **Path** możemy również utworzyć wykorzystując klasę **FileSystems**

```
Path path1 = FileSystems.getDefault().getPath("pandas/cuddly.png");

Path path2 = FileSystems.getDefault().getPath("c:","zooinfo","November",
   "employees.txt");

Path path3 = FileSystems.getDefault().getPath("/home/zoodirector");
```

JAVA.IO.FILE VS JAVA.NIO.FILE.PATH

- W klasie **java.io.File** została udostępniona metoda **toPath** umożliwiająca nam konwersję do typu **java.nio.file.Path**

```
File file = new File("pandas/cuddly.png");
Path path = file.toPath();
```

- Celem zachowania kompatybilności wstecznej, interfejs **java.nio.file.Path** zawiera metodę **toFile()** umożliwiającą konwersję do typu **java.io.File**

```
Path path = Paths.get("cuddly.png");
File file = path.toFile();
```

ŚCIEŻKA BEZWZGLĘDNA
ŚCIEŻKA WZGLĘDNA

Reguły obowiązujące w kontekście egzaminu:

- Jeżeli ścieżka rozpoczyna się od znaku: „/", np.: „/bird/parrot" – jest to ścieżka **bezwzględna**
- Jeżeli ścieżka rozpoczyna się literą dysku, np.: „C:\bird\emu" – jest to ścieżka **bezwzględna**
- W pozostałych przypadkach – mamy do czynienia ze ścieżką **względną**, np.: „..\eagle"

PATH - WŁAŚCIWOŚCI

Interfejs Path posiada trzy metody umożliwiające uzyskanie podstawowych informacji na temat ścieżki: **toString()**, **getName()**, **getNameCount()**

```
Path path = Paths.get("/land/hippo/harry.happy");
System.out.println("The Path Name is: "+path);

for(int i=0; i<path.getNameCount(); i++) {
   System.out.println("   Element "+i+" is: "+path.getName(i));
}
```

PATH – WŁAŚCIWOŚCI SKŁADOWYCH

Interfejs Path posiada bardzo dużą liczbę metod umożliwiającą uzyskanie informacji na temat składowych danej ścieżki. W szczególności są to: **getFileName()**, **getParent()**, **getRoot()**

```
import java.nio.file.*;

public class PathFilePathTest {
   public static void printPathInformation(Path path) {

      System.out.println("Filename is: "+path.getFileName());
      System.out.println("Root is: "+path.getRoot());

      Path currentParent = path;
      while((currentParent = currentParent.getParent()) != null) {
         System.out.println("   Current parent is: "+currentParent);
     }
   }

   public static void main(String[] args) {
      printPathInformation(Paths.get("/zoo/armadillo/shells.txt"));
      System.out.println();
      printPathInformation(Paths.get("armadillo/shells.txt"));
   }
}
```

PATH – ŚCIEŻKA BEZWZGLĘDNA?

Interfejs Path posiada dwie metody umożliwiające weryfikację oraz konwersję do postaci ścieżki bezwzględnej. Są to: **isAbsolute()** oraz **toAbsolutePath**

```
Path path1 = Paths.get("C:\\birds\\egret.txt");
System.out.println("Path1 is Absolute? "+path1.isAbsolute());
System.out.println("Absolute Path1: "+path1.toAbsolutePath());

Path path2 = Paths.get("birds/condor.txt");
System.out.println("Path2 is Absolute? "+path2.isAbsolute());
System.out.println("Absolute Path2 "+path2.toAbsolutePath());
```

PATH – KREACJA NOWEJ WZGLĘDNEJ ŚCIEŻKI

Interfejs Path udostępnia metodę **subpath(int, int)** umożliwiającą utworzenie nowej względnej ścieżki na bazie istniejącej.

```
Path path = Paths.get("/mammal/carnivore/raccoon.image");
System.out.println("Path is: "+path);

System.out.println("Subpath from 0 to 3 is: "+path.subpath(0,3));
System.out.println("Subpath from 1 to 3 is: "+path.subpath(1,3));
System.out.println("Subpath from 1 to 2 is: "+path.subpath(1,2));
```

```
Path is: /mammal/carnivore/raccoon.image
Subpath from 0 to 3 is: mammal/carnivore/raccoon.image
Subpath from 1 to 3 is: carnivore/raccoon.image
Subpath from 1 to 2 is: carnivore
```

```
System.out.println("Subpath from 0 to 4 is: "+path.subpath(0,4)); // THROWS
                                                    // EXCEPTION AT RUNTIME

System.out.println("Subpath from 1 to 1 is: "+path.subpath(1,1)); // THROWS
                                                    // EXCEPTION AT RUNTIME
```

PATH – RELATIVIZE

Interfejs Path udostępnia metodę **relativize(Path)** umożliwiającą utworzenie nowej **względnej ścieżki** pomiędzy poszczególnymi obiektami **Path**.

```
Path path1 = Paths.get("fish.txt");
Path path2 = Paths.get("birds.txt");
System.out.println(path1.relativize(path2));
System.out.println(path2.relativize(path1));
```

```
..\birds.txt
..\fish.txt
```

```
Path path3 = Paths.get("E:\\habitat");
Path path4 = Paths.get("E:\\sanctuary\\raven");
System.out.println(path3.relativize(path4));
System.out.println(path4.relativize(path3));
```

```
..\sanctuary\raven
..\..\habitat
```

```
Path path1 = Paths.get("/primate/chimpanzee");
Path path2 = Paths.get("bananas.txt");
Path1.relativize(path3); // THROWS EXCEPTION AT RUNTIME
```

```
Path path3 = Paths.get("c:\\primate\\chimpanzee");
Path path4 = Paths.get("d:\\storage\\bananas.txt");
path3.relativize(path4); // THROWS EXCEPTION AT RUNTIME
```

PATH – RESOLVE

Interfejs Path udostępnia metodę **resolve(Path)** umożliwiającą utworzenie nowej ścieżki poprzez połączenie (join) istniejącej ścieżki ze ścieżką bieżącą.

```
final Path path1 = Paths.get("/cats/../panther");
final Path path2 = Paths.get("food");
System.out.println(path1.resolve(path2));
```

```
/cats/../panther/food
```

```
final Path path1 = Paths.get("/turkey/food");
final Path path2 = Paths.get("/tiger/cage");
System.out.println(path1.resolve(path2));
```

```
/tiger/cage
```

PATH – NORMALIZE

Metoda relativize() umożliwia tworzenie **ścieżek względnych** uwzględniających w swojej konstrukcji odwołania „**.**” oraz „**..**”. W celu eliminacji potencjalnych nadmiarowych (zbędnych) wystąpień ww. konstrukcji, możemy posłużyć się metodą **normalize()**.

```
Path path3 = Paths.get("E:\\data");
Path path4 = Paths.get("E:\\user\\home");

Path relativePath = path3.relativize(path4);
System.out.println(path3.resolve(relativePath));
```

```
E:\data\..\user\home
```

```
System.out.println(path3.resolve(relativePath).normalize());
```

```
E:\user\home
```

PATH – WERYFIKACJA ISTNIENIA PLIKU

Metoda toRealPath(Path) przyjmuje jako parametr wywołania obiekt Path i zwraca referencję do rzeczywistej ścieżki w zakresie systemu plików. W swoim zachowaniu jest zbieżna z metodą **toAbsolutePath** (konwersja ścieżki względnej do ścieżki bezwzględnej), ale dodatkowo rzuca wyjątek **IOException** w sytuacji, w której dany plik nie istnieje. Dodatkowo, w wyniku tego wywołania, uzyskiwana ścieżka jest znormalizowana. Metoda **toRealPath**(Path) wspiera obsługę linków symbolicznych. Pobranie bieżącego katalogu (**cwd**) można zrealizować:

```
System.out.println(Paths.get(".").toRealPath());
```

```
/zebra/food.source → /horse/food.txt
```

```
try {
   System.out.println(Paths.get("/zebra/food.source").toRealPath());

   System.out.println(Paths.get(".././food.txt").toRealPath());
} catch (IOException e) {
   // Handle file I/O exception...
}
```

```
/horse/food.txt
/horse/food.txt
```

FILES - INTERAKCJA

- Metoda **Files.exists(Path)** zwraca wartość **true** wtedy i tylko wtedy, gdy plik / katalog, na który referuje ścieżka istnieje.

```
Files.exists(Paths.get("/ostrich/feathers.png"));

Files.exists(Paths.get("/ostrich"));
```

- Metoda **Files.isSameFile(Path, Path)** weryfikuje, czy obydwa obiekty typu Path wskazują na ten sam plik. Metoda ta nie porównuje zawartości plików! Metoda uwzględnia linki symboliczne.

```
try {
   System.out.println(Files.isSameFile(Paths.get("/user/home/cobra"),
      Paths.get("/user/home/snake")));

   System.out.println(Files.isSameFile(Paths.get("/user/tree/../monkey"),
      Paths.get("/user/monkey")));

   System.out.println(Files.isSameFile(Paths.get("/leaves/./giraffe.exe"),
      Paths.get("/leaves/giraffe.exe")));

   System.out.println(Files.isSameFile(Paths.get("/flamingo/tail.data"),
      Paths.get("/cardinal/tail.data")));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

- Metody **createDirectory()** oraz **createDirectories()** (podobnie jak metody mkdir() oraz mkdirs() udostępniane na klasie File) umożliwiają tworzenie katalogów.

```
try {
   Files.createDirectory(Paths.get("/bison/field"));

   Files.createDirectories(Paths.get("/bison/field/pasture/green"));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

- Metoda **Files.copy(Path, Path)** umożliwia kopiowanie zawartości pliku. Domyślnie, metoda uwzględnia linki symboliczne, nie nadpisuje pliku lub katalogu jeżeli takowy już istnieje oraz nie kopiuje atrybutów danego pliku.

```
try {
   Files.copy(Paths.get("/panda"), Paths.get("/panda-save"));

   Files.copy(Paths.get("/panda/bamboo.txt"),
      Paths.get("/panda-save/bamboo.txt"));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

```
try (InputStream is = new FileInputStream("source-data.txt");
     OutputStream out = new FileOutputStream("output-data.txt")) {

   // Copy stream data to file
   Files.copy(is, Paths.get("c:\\mammals\\wolf.txt"));

   // Copy file data to stream
   Files.copy(Paths.get("c:\\fish\\clown.xsl"), out);

} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

- Metoda **Files.move(Path, Path)** umożliwia zmianę lokalizacji pliku lub zmianę jego nazwy. Domyślnie, operacja ta uwzględnia linki symboliczne, rzuca wyjątkiem w sytuacji, gdy docelowy plik już istnieje. Ponadto, operacja ta nie jest (domyślnie) operacją atomową.

```
try {
   Files.move(Paths.get("c:\\zoo"), Paths.get("c:\\zoo-new"));

   Files.move(Paths.get("c:\\user\\addresses.txt"),
      Paths.get("c:\\zoo-new\\addresses.txt"));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

Metody **Files.delete(Path)** oraz **Files.deleteIfExists(Path)** umożliwiają usunięcie pliku lub katalogu. W sytuacji, w której danej pliku nie istnieje, metoda **delete(Path)** rzuci wyjątek. Próba usunięcia katalogu, który nie jest katalogiem pustym kończy się wyjątkiem: **DirectoryNotEmptyException**. W sytuacji, w której obiekt Path jest linkiem symboliczny, **nie jest** usuwany plik na który ten link wskazuje, a jedynie sam link symboliczny.

```
try {
   Files.delete(Paths.get("/vulture/feathers.txt"));
   Files.deleteIfExists(Paths.get("/pigeon"));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

Klasa **Files** udostępnia ponadto metody **newBufferedReader(Path, Charset)** oraz **newBufferedWriter(Path, Charset)** umożliwiające odczyt oraz zapis do pliku.

Uwaga. Domyślny sposób kodowania uzyskujemy poprzez wywołanie: **Charset.defaultCharset()**

```
Path path = Paths.get("/animals/gopher.txt");
try (BufferedReader reader = Files.newBufferedReader(path,
            Charset.forName("US-ASCII"))) {
   // Read from the stream
   String currentLine = null;
   while((currentLine = reader.readLine()) != null)
      System.out.println(currentLine);
} catch (IOException e) {
   // Handle file I/O exception...
}
```

```
Path path = Paths.get("/animals/gorilla.txt");
List<String> data = new ArrayList();
try (BufferedWriter writer = Files.newBufferedWriter(path,
         Charset.forName("UTF-16"))) {
   writer.write("Hello World");
} catch (IOException e) {
   // Handle file I/O exception...
}
```

FILES - INTERAKCJA

Klasa **Files** udostępnia metodę umożliwiającą odczyt wszystkich linii w danym pliku.

```
Path path = Paths.get("/fish/sharks.log");
try {
   final List<String> lines = Files.readAllLines(path);
   for(String line: lines) {
      System.out.println(line);
   }
} catch (IOException e) {
   // Handle file I/O exception...
}
```

WŁAŚCIWOŚCI PLIKÓW

Klasa **Files** udostępnia trzy metody umożliwiające weryfikację, czy dana ścieżka pliku referuje do:

- katalogu – **Files.isDirectory(Path)**
- pliku – **Files.isRegularFile(Path)**
- linku symbolicznego – **Files.isSymbolicLink(Path)**

	isDirectory()	isRegularFile()	isSymbolicLink()
/canine/coyote	true	false	false
/canine/types.txt	false	true	false
/coyotes	true if the target is a directory	true if the target is a regular file	true

WŁAŚCIWOŚCI PLIKÓW

Pozostałe metody udostępniane w klasie **Files**:

- **isHidden(Path)**

```
try {
   System.out.println(Files.isHidden(Paths.get("/walrus.txt")));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

- **isReadable(Path)**, **isExecutable(Path)**
- **size(Path)**

```
try {
   System.out.println(Files.size(Paths.get("/zoo/c/animals.txt")));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

WŁAŚCIWOŚCI PLIKÓW

- **getLastModifiedTime(Path), setLastModifiedTime(Path, FileTime)**

```
try {
   final Path path = Paths.get("/rabbit/food.jpg");
   System.out.println(Files.getLastModifiedTime(path).toMillis());

   Files.setLastModifiedTime(path,
            FileTime.fromMillis(System.currentTimeMillis()));

   System.out.println(Files.getLastModifiedTime(path).toMillis());
} catch (IOException e) {
   // Handle file I/O exception...
}
```

- **getOwner(Path), setOwner(Path, UserPrincipal)**

```
try {
   // Read owner of file
   Path path = Paths.get("/chicken/feathers.txt");
   System.out.println(Files.getOwner(path).getName());

   // Change owner of file
   UserPrincipal owner = path.getFileSystem()
         .getUserPrincipalLookupService().lookupPrincipalByName("jane");
   Files.setOwner(path, owner);

   // Output the updated owner information
   System.out.println(Files.getOwner(path).getName());
} catch (IOException e) {
   // Handle file I/O exception...
}
```

WIDOKI WŁAŚCIWOŚCI PLIKÓW

Attributes Class	View Class	Description
`BasicFileAttributes`	`BasicFileAttributeView`	Basic set of attributes supported by all file systems
`DosFileAttributes`	`DosFileAttributeView`	Attributes supported by DOS/ Windows-based systems
`PosixFileAttributes`	`PosixFileAttributeView`	Attributes supported by POSIX systems, such as UNIX, Linux, Mac, and so on

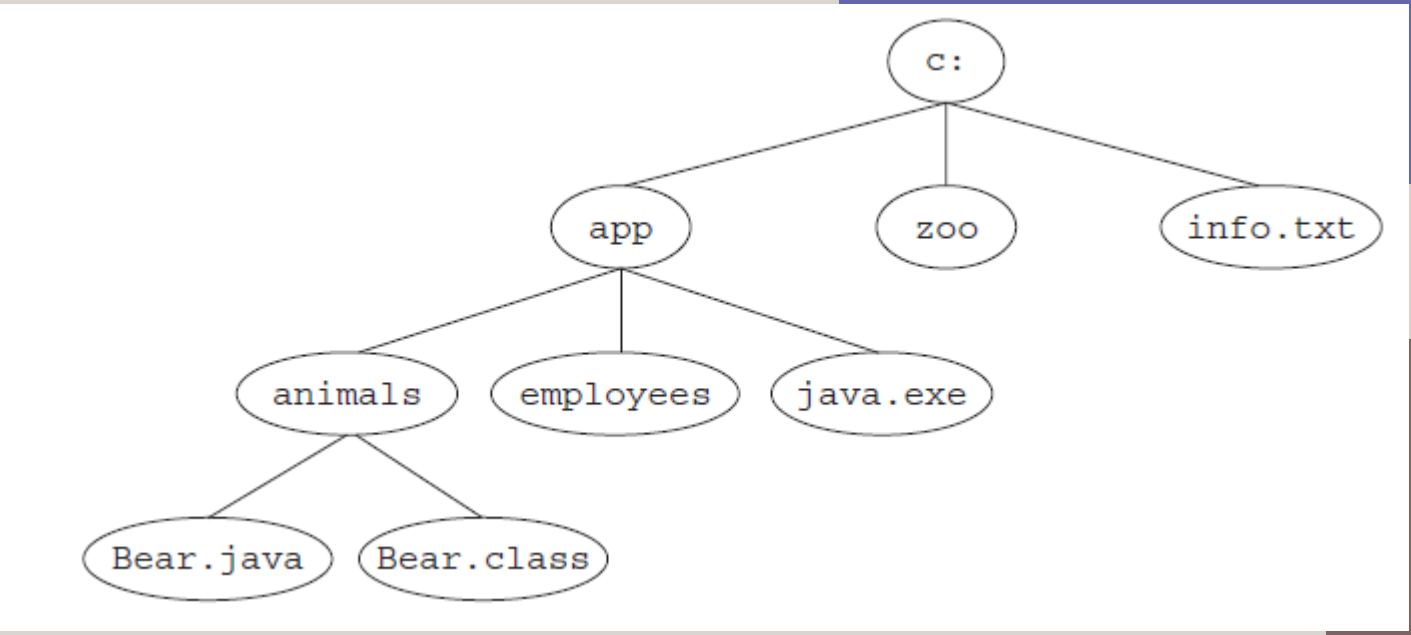
WIDOKI WŁAŚCIWOŚCI PLIKÓW - ODCZYT

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

public class BasicFileAttributesSample {
   public static void main(String[] args) throws IOException {
      Path path = Paths.get("/turtles/sea.txt");
      BasicFileAttributes data = Files.readAttributes(path,
         BasicFileAttributes.class);

      System.out.println("Is path a directory? "+data.isDirectory());
      System.out.println("Is path a regular file? "+data.isRegularFile());
      System.out.println("Is path a symbolic link? "+data.isSymbolicLink());
      System.out.println("Path not a file, directory, nor symbolic link? "+
         data.isOther());

      System.out.println("Size (in bytes): "+data.size());

      System.out.println("Creation date/time: "+data.creationTime());
      System.out.println("Last modified date/time: "+data.lastModifiedTime());
      System.out.println("Last accessed date/time: "+data.lastAccessTime());
      System.out.println("Unique file identifier (if available): "+
         data.fileKey());
   }
}
```

WIDOKI WŁAŚCIWOŚCI PLIKÓW - MODYFIKACJA

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;

public class BasicFileAttributeViewSample {
   public static void main(String[] args) throws IOException {
      Path path = Paths.get("/turtles/sea.txt");

      BasicFileAttributeView view =
         Files.getFileAttributeView(path,BasicFileAttributeView.class);
      BasicFileAttributes data = view.readAttributes();

      FileTime lastModifiedTime = FileTime.fromMillis(
         data.lastModifiedTime().toMillis()+10_000);

      view.setTimes(lastModifiedTime,null,null);
   }
}
```

PRZESZUKIWANIE DRZEWA PLIKÓW



PRZESZUKIWANIE PLIKÓW

Przeszukiwanie drzewa plików (w głąb – **D**epth **F**irst **S**earch) jest realizowane w oparciu o metodę **Files.walk(Path)**

```
Path path = Paths.get("/bigcats");

try {
   Files.walk(path)
      .filter(p -> p.toString().endsWith(".java"))
      .forEach(System.out::println);
} catch (IOException e) {
   // Handle file I/O exception...
}
```

PRZESZUKIWANIE PLIKÓW

Proces wyszukiwania plików lub katalogów spełniających określone kryteria można zrealizować w oparciu o metodę **Files.find(Path, BasicFileAttributes)**.

```
Path path = Paths.get("/bigcats");
long dateFilter = 1420070400000l;

try {
   Stream<Path> stream = Files.find(path, 10,
     (p,a) -> p.toString().endsWith(".java")
       && a.lastModifiedTime().toMillis()>dateFilter);
   stream.forEach(System.out::println);
} catch (Exception e) {
   // Handle file I/O exception...
}
```

PRZESZUKIWANIE PLIKÓW

Listowanie zawartości katalogu realizujemy przy pomocy metody **Files.list(Path)**.

```
Path path = Paths.get("ducks");
Files.list(path)
   .filter(p -> !Files.isDirectory(p))
   .map(p -> p.toAbsolutePath())
   .forEach(System.out::println);
```

WYPISYWANIE ZAWARTOŚCI PLIKU

Zawartość pliku (oprócz poznanych dotychczas sposobów) możemy wypisać wykorzystując metodę **Files.lines(Path)**.

```
Path path = Paths.get("/fish/sharks.log");
try {
   Files.lines(path).forEach(System.out::println);
} catch (IOException e) {
   // Handle file I/O exception...
}
```

Przykład:

```
Path path = Paths.get("/fish/sharks.log");
try {
   System.out.println(Files.lines(path)
      .filter(s -> s.startsWith("WARN "))
      .map(s -> s.substring(5))
      .collect(Collectors.toList()));
} catch (IOException e) {
   // Handle file I/O exception...
}
```

DZIĘKUJĘ ZA UWAGĘ