Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

JAVA FOUNDATIONS: EXAM NUMBER: 1Z0-811

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ZAKRES SZKOLENIA – DZIEŃ 3

- **Wejście / wyjście (Java IO, NIO.2)**
- **Obsługa daty i czasu w Java SE 8**

  LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration

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)

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
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

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.

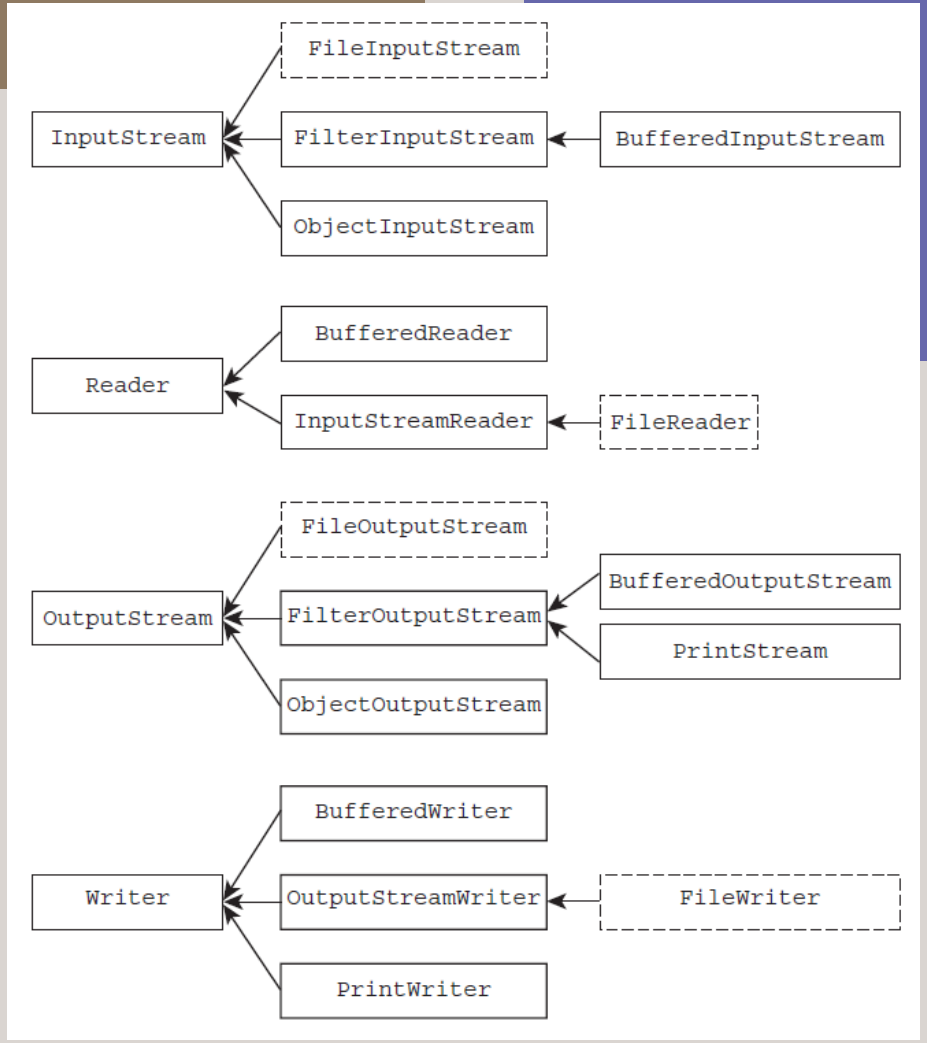
WYBRANE METOD KLASY FILE

`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

```
1  File file = new File("C:\\data\\zoo.txt");
2  System.out.println("File exists: " + file.exists());;
3  if (file.exists()) {
4      System.out.println("Absolute path: " + file.getAbsolutePath());
5      System.out.println("Is directory: " + file.isDirectory());
6      System.out.println("Parent path: " + file.getParent());
7
8      if (file.isFile()) {
9          System.out.println("File size: " + file.length());
10         System.out.println("File last modified at: " + file.lastModified());
11     } else {
12         for (File subFile : file.listFiles()) {
13             System.out.println("\t" + subFile.getName());
14         }
15     }
16 }
```

KLASY PAKIETU JAVA.IO



Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

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);
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

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"));
      }
   }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

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

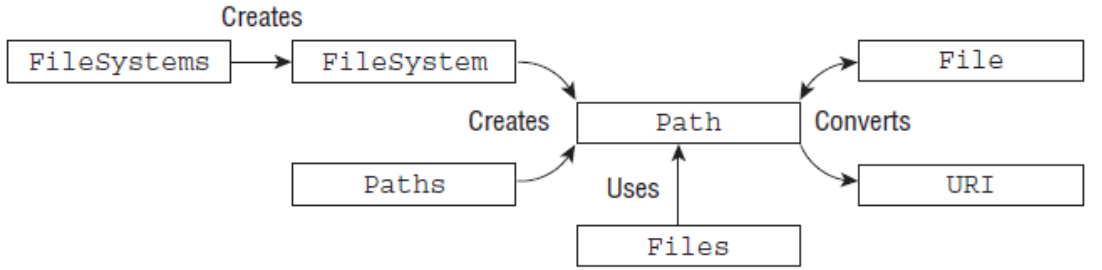
- Wykorzystanie klasy **Paths**

```
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
```

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

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

- 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 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

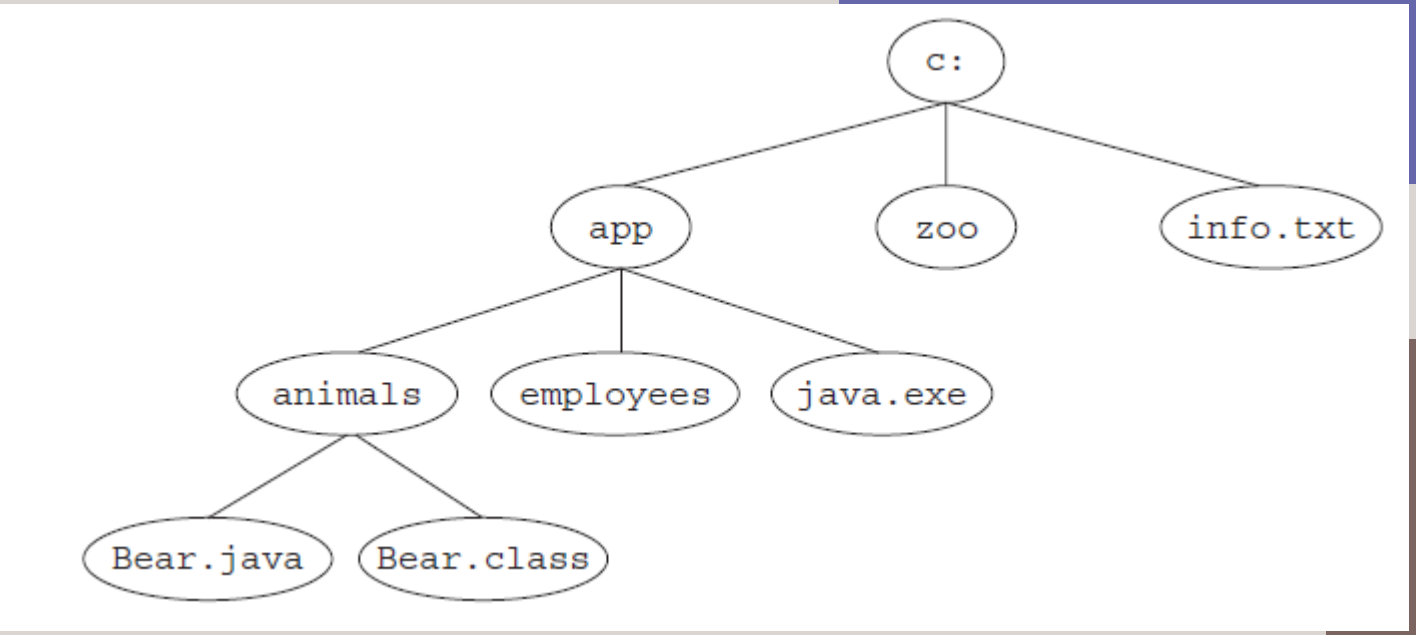
Klasa **Files** udostępnia trzy metody umożliwiające weryfikację, czy dana ścieżka pliku referuje do:

- 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

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

PRZESZUKIWANIE DRZEWA PLIKÓW



Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

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...
}
```

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

DZIĘKUJĘ ZA UWAGĘ