

The background is a solid blue gradient. Overlaid on this are numerous thin, white, curved lines that flow from the left side towards the right, creating a sense of motion and depth. These lines are more densely packed in some areas, forming peaks and valleys, similar to a stylized wave or a series of overlapping ridges.

JAVA SE 8 PROGRAMMER II

ZAKRES SZKOLENIA – DZIEŃ 4

- API dot. Daty / czasu w Javie SE 8
- Podstawy Java IO
- Plikowe I/O w Javie (NIO.2)
- Wielowątkowość
- Tworzenie aplikacji z wykorzystaniem JDBC
- Tworzenie aplikacji z wykorzystaniem JDBC



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 LocalDateTime now = LocalDateTime.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
```

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

```
1 // aktualna z domyślną strefą
2 ZonedDateTime zonedDateTime1 = ZonedDateTime.now();
3 System.out.println("Aktualny czas w strefie komputera (serwera): " + zonedDateTime1);
4
5 // aktualna w konkretnej strefie
6 ZonedDateTime zonedDateTime2 = ZonedDateTime.now(ZoneId.of("Poland"));
7 System.out.println("Aktualny czas w strefie czasowej Polski: " + zonedDateTime2);
8
9 // na podstawie LocalDateTime w konkretnej strefie
10 LocalDateTime localDateTime = LocalDateTime.parse("2015-05-30T14:20:50");
11 ZonedDateTime zonedDateTime3 = ZonedDateTime.of(localDateTime, ZoneId.of("Japan"));
12 System.out.println("Data '2015-05-30T14:20:50' w strefie czasowej Japoni: " + zonedDateTime3);
```

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);
```

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
```

INSTANT

Klasa ta wyróżnia się tym, że reprezentuje konkretny i **jednoznacznie określony punkt w czasie** (z **dokładnością do nanosekund**). 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.



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");
```



WYBRANE METODY KLASY FILE

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



WYBRANE METODY KLASY FILE - PRZYKŁAD

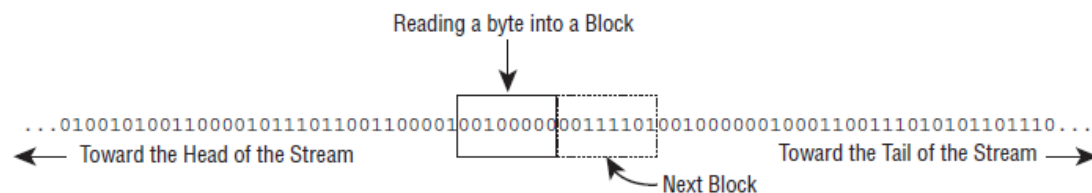
```
1 File file = new File("C:\\data\\zoo.txt");
2 System.out.println("File exists: " + file.exists());
3 if (file.exists()) {
4     System.out.println("Absolute path: " + file.getAbsolutePath());
5     System.out.println("Is directory: " + file.isDirectory());
6     System.out.println("Parent path: " + 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 }
```

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**
(charakterystyczne dla strumieni „buffered”)



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());  
    }  
)
```

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 **niebuforowanych** 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;
    }

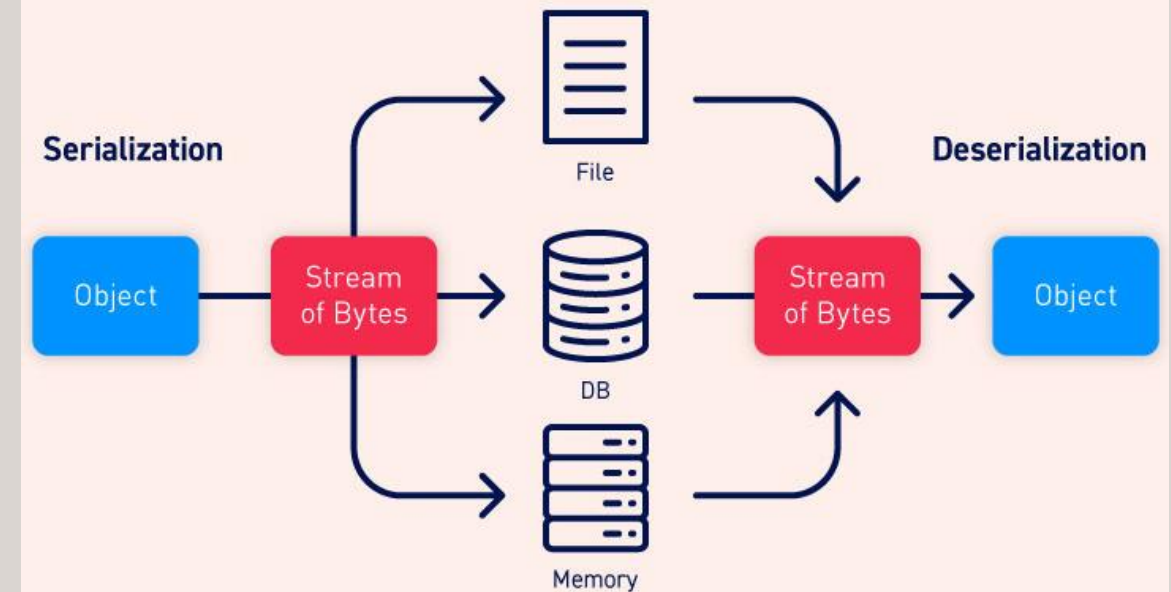
    public static void writeFile(List<String> data, File destination) throws
        IOException {
        try (BufferedWriter writer = new BufferedWriter(
            new FileWriter(destination))) {

            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);
    }
}
```

SERIALIZACJA ORAZ DESERIALIZACJA

- **Serializacja** jest mechanizmem, który realizuje konwersję **stanu obiektu na strumień bajtów**. **Deserializacja** jest natomiast procesem odwrotnym, który umożliwia **odtworzenie 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

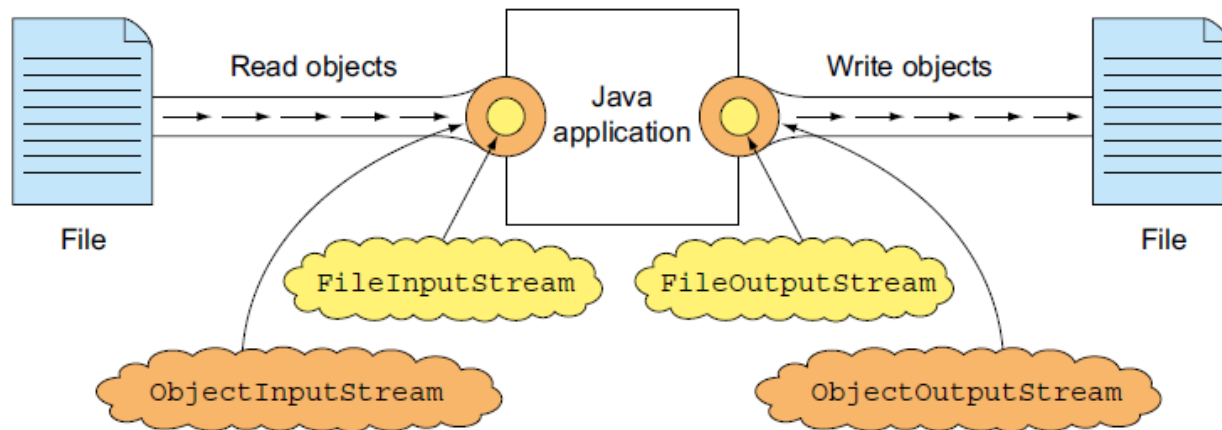
- UWAGI

- 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 *serialVersionUID* – 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 ObjectOutputStreamSample {

    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  
    ...  
}
```

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).**



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
- Konceptyjny **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 - 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");
```

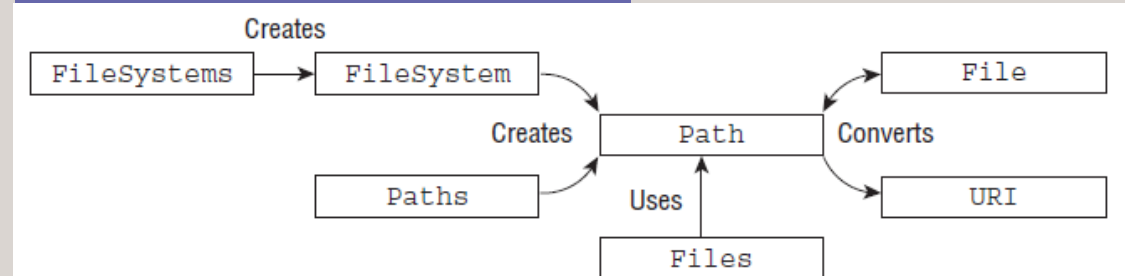
```
Path path2 = Paths.get("c:","zooinfo","November","employees.txt");
```

```
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

- 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();
```



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 – 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));
```

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

E:\data\..\user\home

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

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...
}
```



WIELOWĄTKOWOŚĆ

Programowanie współbieżne obejmuje projektowanie i tworzenie programów, które w fazie wykonania składają się **z co najmniej dwóch jednostek wykonywanych współbieżnie** (każda z nich jest procesem sekwencyjnym), wraz z zapewnieniem synchronizacji pomiędzy nimi. Jednostkami takimi mogą być wątki, bądź też procesy.

THREAD

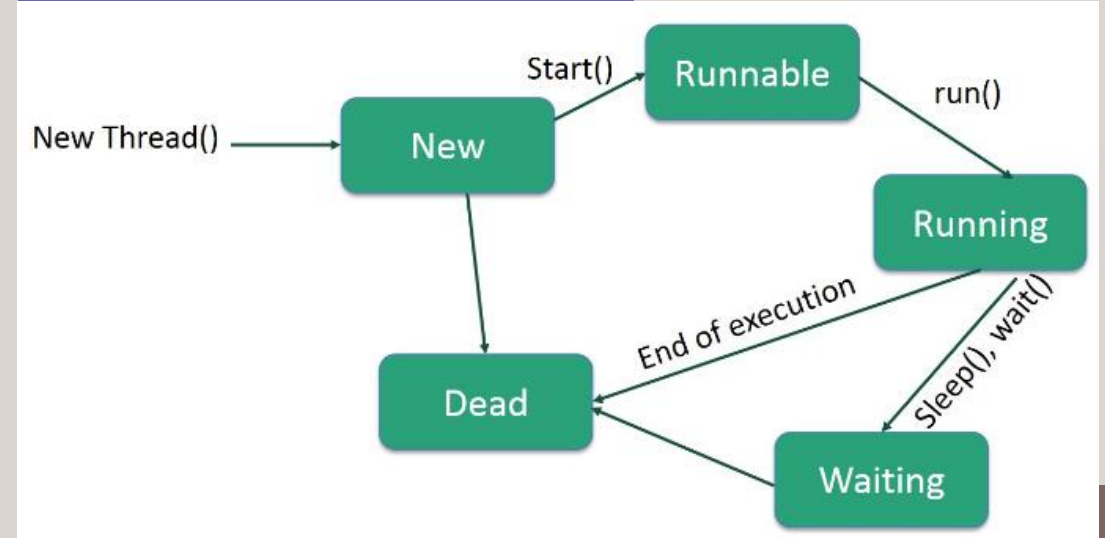
Thread jest **wątkiem wykonania w programie**.

Wirtualna maszyna Java pozwala aplikacji na jednoczesne działanie wielu wątków. **Każdy wątek ma priorytet**.

Wątki o wyższym priorytecie są wykonywane przed wątkami o niższym priorytecie.

Kiedy JVM startuje, zazwyczaj wykonywany jest **wątek główny, który wywołuje metodę main**. Wątki te pracują tak długo, aż nastąpi jeden z poniższych przypadków:

- wątki skończą swoją pracę
- wątek rzuci wyjątek
- nastąpi wywołanie metody **System.exit()** (z dowolnego wątku).



Constant Variable	Value
Thread.MIN_PRIORITY	1
Thread.NORM_PRIORITY	5
Thread.MAX_PRIORITY	10

TWORZENIE WĄTKÓW

Wątek w Javie reprezentowany jest przez klasę **Thread**.

Wątki możemy tworzyć na wiele sposobów, np.:

- rozszerzając klasę **Thread** i **nadpisując metodę run**
- **implementując interfejs funkcyjny Runnable**



DZIEDZICZENIE PO KLASIE THREAD

- Identyfikator wątku można pobrać poprzez wywołanie:
Thread.currentThread().getId()
- UWAGA: Dziedziczenie po klasie Thread jest **NIEZALECANE**. Chcąc stworzyć nowy wątek skorzystajmy z interfejsu Runnable.

```
public class ThreadsExample {  
    public static void main(String[] args) {  
        new HelloWorldThread().start();  
        System.out.println(Thread.currentThread().getId());  
    }  
}  
  
class HelloWorldThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Hello World from another Thread");  
        System.out.println(Thread.currentThread().getId());  
    }  
}
```

RUNNABLE

Innym, lepszym sposobem utworzenia wątku jest zadeklarowanie klasy, **która implementuje interfejs Runnable, z jedną, abstrakcyjną, bezargumentową metodą run**. Instancję Runnable można przekazać jako **argument konstruktora klasy Thread**. Tak stworzony wątek startujemy za **pomocą metody start**.

```
public class ThreadsExample {
    public static void main(String[] args) {
        new Thread(new HelloWorldRunnableThread()).start();
        new Thread(() -> System.out.println(
            "Hello from another thread implemented with lambda")).start();
    }
}

class HelloWorldRunnableThread implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello world from another Thread");
    }
}
```

SYNCHRONIZACJA

Język JAVA wprowadza dwa podstawowe sposoby synchronizacji:

- **synchronizacja metody**
- **synchronizacja bloku kodu**

```
class DummyPairIncrementer implements Runnable {
    private final Pair pair;

    public DummyPairIncrementer(final Pair pair) {
        this.pair = pair;
    }

    @Override
    public void run() {
        for (int idx = 0; idx < 100; idx++) {
            pair.incrementLeft();
            pair.incrementRight();
        }
        System.out.println(pair.getLeft() + " " + pair.getRight());
    }
}
```

```
class Pair {
    private Integer left;
    private Integer right;

    public Pair(final Integer left, final Integer right) {
        this.left = left;
        this.right = right;
    }

    public void incrementLeft() {
        left++;
    }

    public void incrementRight() {
        right++;
    }

    public Integer getLeft() {
        return left;
    }

    public Integer getRight() {
        return right;
    }
}
```

```
public class ThreadsExample {
    public static void main(String[] args) {
        final Pair pair = new Pair(0, 0);
        new Thread(new DummyPairIncrementer(pair)).start();
        new Thread(new DummyPairIncrementer(pair)).start();
    }
}
```

SYNCHRONIZACJA METODY

W celu synchronizacji metody, **do jej deklaracji dodajemy słowo kluczowe synchronized.**

Synchronizacja metody polega na tym, że wątek, który ją wywołuje, **ma do niej wyłączny dostęp do czasu jej zakończenia.** Aby poprawić problem w poprzedniego przykładu, musimy zsynchronizować następujące metody:

```
public synchronized void incrementLeft() {  
    left++;  
}  
  
public synchronized void incrementRight() {  
    right++;  
}
```



SYNCHRONIZACJA BLOKU

Synchronizacja bloku **realizuje dokładnie ten sam mechanizm**, co synchronizacja metody, natomiast **może redukować zakres synchronizacji danych, tylko do poszczególnych instrukcji** związanych, np. z polem klasy. W celu realizacji synchronizacji blokowej również wykorzystywane jest słowo kluczowe **synchronized**, ale dodatkowo **pomiędzy () wstawiamy obiekt, do którego chcemy dostawać się w ramach współbieżności w sposób sekwencyjny**.

```
public void incrementLeft() {  
    System.out.println("Out of synchronized block");  
    synchronized (this) {  
        left++;  
        System.out.println("In synchronized block");  
    }  
    System.out.println("Out of synchronized block");  
}  
  
public void incrementRight() {  
    System.out.println("Out of synchronized block");  
    synchronized (this) {  
        right++;  
        System.out.println("In synchronized block");  
    }  
    System.out.println("Out of synchronized block");  
}
```

DEADLOCK

W sytuacji, w której **kilka wątków blokuje się wzajemnie w nieskończoność**, mamy do czynienia z sytuacją zwaną **zakleszczaniem**. Program nie jest w stanie zakończyć wskazanej operacji, ze względu na trwałą i wzajemną blokadę zasobów. Można ją opisać w następujący sposób:

- **A czeka na B**, ponieważ:
- **B czeka na A**

```
public class DeadLockExample {
    public static void main(String[] args) throws InterruptedException {
        final String r1 = "r1";
        final String r2 = "r2";

        Thread t1 = new Thread() {
            public void run() {
                synchronized (r1) {
                    System.out.println("Thread 1: Locked r1");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ignored) {}
                }
                synchronized (r2) {
                    System.out.println("Thread 1: Locked r2");
                }
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                synchronized (r2) {
                    System.out.println("Thread 2: Locked r1");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ignored) {}
                }
                synchronized (r1) {
                    System.out.println("Thread 2: Locked r2");
                }
            }
        };

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Exiting? No I will never reach this line "
            + "of code because threads will NOT join");
    }
}
```

STARVATION

Zagłodzenie jest stanem, w którym jeden z wątków nie jest w stanie ukończyć swojego zadania ze względu na permanentny brak dostępu do określonego zasobu. Problem ten może wynikać z ustalonego niskiego priorytetu danego wątku. Problem ten można rozwiązać na poziomie algorytmu.



LIVELOCK

Livelock stanowi specjalny przypadek zagłódzenia.

Występuje gdy obydwie wątki, aby uniknąć deadlock'a zatrzymują wykonywanie kodu, aby dać szansę innym wątkom na wykonanie się. Livelock może wydawać się podobny do deadlock (rezultat jest taki sam).

W przypadku Livelock stan procesu się jednak zmienia. Z kolei w deadlock, pozostaje ciągle taki sam.



CALLABLE

Generyczny interfejs funkcyjny reprezentujący zadanie, które **może zwracać albo rezultat**, albo wyjątek za pomocą **metody bezargumentowej call()**. Interfejs Callable jest zbliżony do interfejsu Runnable z tą różnicą, że **metoda run nie może zwracać żadnego rezultatu**. Obydwa interfejsy są do siebie podobne ze względu na potencjalne zastosowanie w obsłudze wielowątkowej.

```
public class GetRequest implements Callable<String> {  
  
    @Override  
    public String call() throws Exception {  
        return "Dummy http response";  
    }  
}
```

EXECUTORSERVICE

Tworząc aplikacje wielowątkowe rzadko wykorzystujemy niskopoziomowe API i ręcznie zarządzamy wątkami. W miarę możliwości powinniśmy korzystać z tzw. puli wątków, czyli grupy wątków zarządzanych przez zewnętrzny byt. Jednym z takich mechanizmów w Javie jest **interfejs ExecutorService**, który **upraszcza wykonywanie zadań w trybie asynchronicznym, wykorzystując do tego pewną pulę wątków**. Aby stworzyć instancję **ExecutorService**, możemy **wykorzystać fabrykę, klasę Executors**:

- **newSingleThreadExecutor()**
- **newFixedThreadPool(int nThreads)**

```
public class ExecutorsCreationExample {  
    public static void main(String[] args) throws InterruptedException {  
        final int cpus = Runtime.getRuntime().availableProcessors();  
        // pula z pojedynczym wątkiem  
        final ExecutorService singleThreadES = Executors.newSingleThreadExecutor();  
        // pula z ilością wątków równą ilości cp  
        final ExecutorService executorService = Executors.newFixedThreadPool(cpus);  
    }  
}
```

EXECUTORSERVICE

- ZAMYKANIE

Tworząc ExecutorService musimy pamiętać o jego ręcznym zamknięciu. Służą do tego następujące metody:

- **shutdown()** - pula wątków przestaje przyjmować nowe zadania, te rozpoczęte zostaną dokończone, a następnie pula zostanie zamknięta
- **shutdownNow()** - podobnie **jak shutdown**, ExecutorService przestanie przyjmować nowe zadania, dodatkowo próbuje zatrzymać wszystkie aktywnie wykonywane zadania, **zatrzymuje przetwarzanie zadań oczekujących i zwraca listę zadań oczekujących na wykonanie.**



WYKONYWANIE ZADAŃ

W celu wykonania zadania na wątku z puli, możemy wykorzystać metody:

- **submit()** - wykonuje zadanie typu **Callable**, bądź też **Runnable**

```
public class CallableFutureExample {
    public static void main(String[] args) {
        // stworzenie ExecutorService z jednowątkową pulą
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        // implementacja Callable za pomocą lambda
        Future<String> result = executorService.submit(() -> "I am result of callable!");
        try {
            System.out.println("Print result of the future: " + result.get());
        } catch (InterruptedException | ExecutionException e) {
            System.err.println("Oops");
        }
        // pamiętajmy o ręcznym zamknięciu ExecutorService
        executorService.shutdown();
    }
}
```


ZMIENNE ATOMOWE

Inkrementacja, z perspektywy języka Java jest pojedynczym wyrażeniem, ale dla procesora **jest to kilka operacji**. Mówimy, że pewna operacja jest **atomowa**, jeżeli podczas jej wykonywania inny wątek nie może przeczytać lub zmienić wartości zmienianych zmiennych. Pakiet **java.util.concurrent.atomic** definiuje klasy, które obsługują operacje atomowe na pojedynczych zmiennych. Odpowiednie typy to:

- **AtomicInteger**
- **AtomicLong**
- **AtomicBoolean**

```
public class AtomicsDemo {
    public static void main(String[] args) {
        final ExecutorService executorService = Executors.newFixedThreadPool(2);
        final AtomicInteger atomicInteger = new AtomicInteger(0);
        executorService.submit(new IncrementingThread(atomicInteger));
        executorService.submit(new IncrementingThread(atomicInteger));
        executorService.shutdown();
    }
}

class IncrementingThread implements Runnable {

    private final AtomicInteger value;

    IncrementingThread(final AtomicInteger value) {
        this.value = value;
    }

    @Override
    public void run() {
        for (int idx = 0; idx < 1000; idx++) {
            value.incrementAndGet();
        }
        // wolniejszy z wątków zawsze wypisze wartość 2000
        System.out.println(value.get());
    }
}
```

STRUMIENIE RÓWNOLEGŁE

Strumień równoległy jest strumieniem umożliwiającym przetwarzanie elementów w sposób równoległy, z wykorzystaniem wielu wątków. Równoległe przetwarzanie strumieni może w niektórych przypadkach znacząco poprawić wydajność.

Strumień równoległy możemy tworzyć w oparciu o:

- metodę **parallel()** wywoływaną na istniejącym strumieniu

```
Stream<Integer> stream = Arrays.asList(1,2,3,4,5,6).stream();  
Stream<Integer> parallelStream = stream.parallel();
```

- metodę **parallelStream()** wywoływaną na kolekcji

```
Stream<Integer> parallelStream2 = Arrays.asList(1,2,3,4,5,6).parallelStream();
```



TWORZENIE APLIKACJI Z WYKORZYSTANIEM JDBC

JDBC (Java DataBase Connectivity) jest to **API**, wbudowane w Javę SE, które pozwala na komunikację z relacyjną bazą danych z poziomu kodu.

Główne funkcjonalności, jakie JDBC realizuje to:

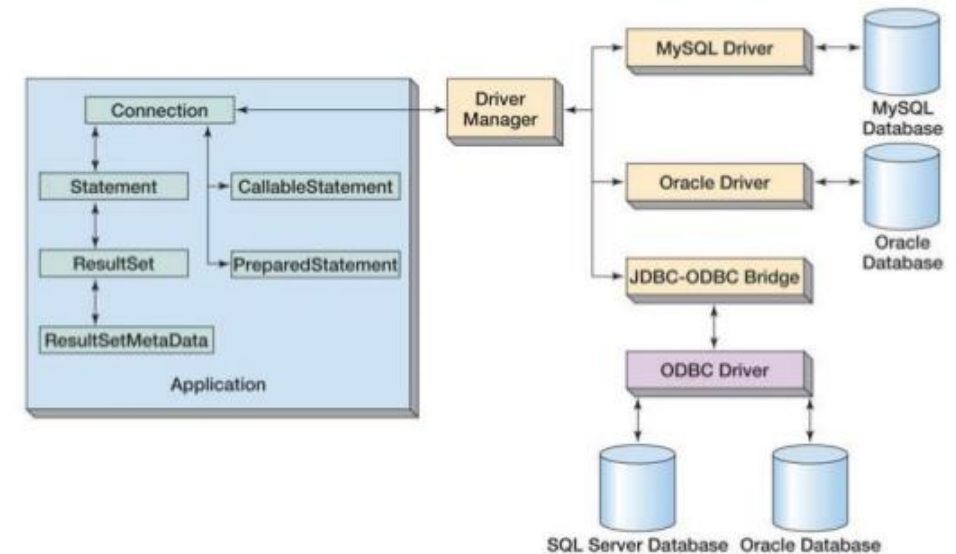
- możliwość **nawiązania połączenia** z bazą danych,
- **wysyłanie zapytań** i instrukcji do bazy danych,
- **pobieranie i przetwarzanie wyników** zapytań SQL.

KOMPONENTY

JDBC składa się z czterech komponentów:

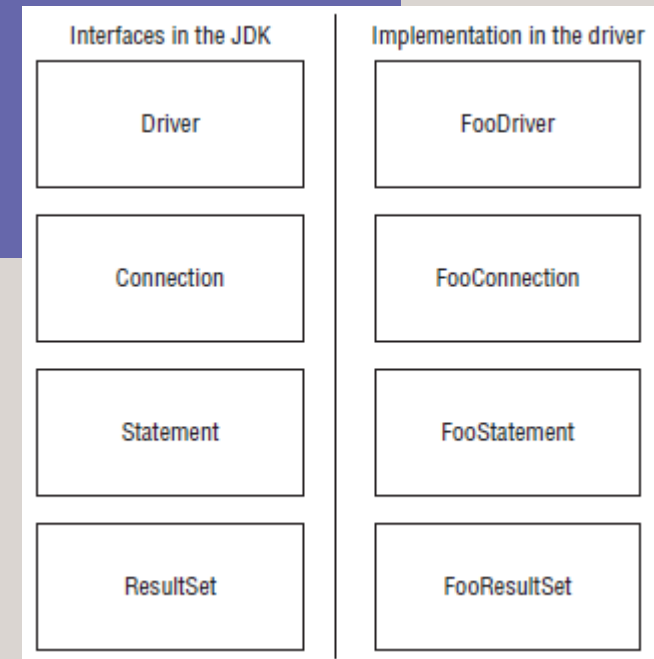
- **JDBC API** - jest to najważniejsza część JDBC, która pozwala, za pomocą wywoływania metod z poziomu kodu, wywoływać zapytania na bazie danych i pobierać ich wyniki. Składniki API znajdziemy w pakietach `java.sql` i `javax.sql`.
- **JDBC Driver Manager** - definiuje w jaki sposób nawiązać połączenie z bazą danych.
- JDBC Test Suite
- JDBC-ODBC Bridge

JDBC Components



KOMPONENTY JDBC

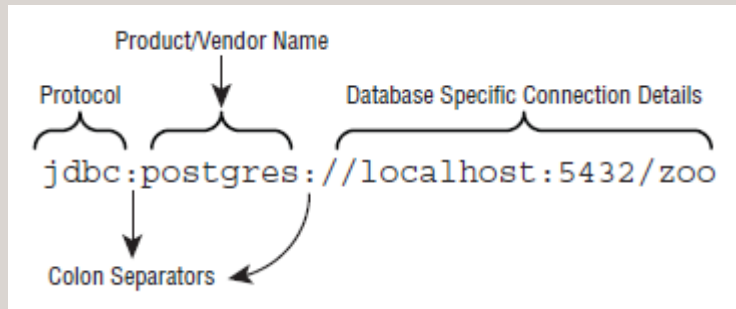
- **Driver** – w jaki sposób uzyskać połączenie z bazą danych
- **Connection** – w jaki sposób komunikować się z bazą danych?
- **Statement** – w jaki sposób wykonywać zapytania SQL?
- **ResultSet** – jakie wyniki zostały zwrócone w wyniku zapytania SELECT?



BUDOWA ADRESU JDBC

Adres **JDBC** składa się z trzech zasadniczych części:

- **protokołu** (JDBC)
- **nazwy produktu / dostawcy bazy danych**
(np. mysql, postgres, oracle)
- **dodatkowych informacji** specyfikujących połączenie z bazą danych (np. lokalizacja serwera, port, nazwa bazy danych, nazwa schematu bazy danych, itd...)



```
jdbc:postgresql://localhost/zoo  
jdbc:oracle:thin:@123.123.123.123:1521:zoo  
jdbc:mysql://localhost:3306/zoo?profileSQL=true
```

```
jdbc:postgresql://local/zoo  
jdbc:mysql://123456/zoo  
jdbc:oracle;thin;/localhost/zoo
```

NAWIAZYWANIE POŁĄCZENIA Z BAZĄ DANYCH

JDBC oferuje **dwa sposoby nawiązywania połączenia** z bazą danych. Umożliwiają to obiekty:

- klasa **DriverManager** wraz z metodą statyczną **getConnection**
- stworzenie implementacji interfejsu **DataSource** i wywołanie metody **getConnection** na instancji klasy

Nieważne, który ze sposobów wybierzemy, do nawiązania połączenia zawsze potrzebujemy pewien zestaw danych.

Są to:

- **nazwa użytkownika**
- **hasło użytkownika**
- **url bazy danych**



NAWIĄZYWANIE POŁĄCZENIA Z BAZĄ DANYCH

W **JDBC w wersji 3.0** oraz wcześniejszych, w celu poprawnego nawiązania połączenia z bazą danych, należało wczytać odpowiednią klasę sterownika.

```
public static void main(String[] args) throws SQLException,
ClassNotFoundException {
    Class.forName("org.postgresql.Driver");
    Connection conn = DriverManager.getConnection(
        "jdbc:postgresql://localhost:5432/ocp-book",
        "username",
        "password");
}
```

Począwszy od wersji **4.0**, wszystkie implementacje sterowników muszą zawierać plik **META-INF/service/java.sql.Driver**

TABLE 10.1 JDBC 3.0 vs. 4.0 drivers

	JDBC <= 3.0 Driver	JDBC >= 4.0 Driver
Required to contain java.sql.Driver	No	Yes
Java will use java.sql.Driver file if present	Yes	Yes
Required to use Class.forName	Yes	No
Allowed to use Class.forName	Yes	Yes

OBIEKT STATEMENT

Interakcja z bazą danych realizowana jest za pomocą obiektu klasy **Statement**, który umożliwia wysyłanie zapytań SQL oraz konsumowanie odpowiedzi.

Obiekt **Statement** jest tworzony w oparciu o instancję **Connection** oraz metodę **createConnection**.

```
Statement stmt = conn.createStatement();
```

```
Statement stmt = conn.createStatement(  
    ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
```



STATEMENT

Mając **nawiązane połączenie do bazy danych** i stworzony obiekt **Statement**, możemy za jego pomocą **wywołać zapytanie SQL**. Do wywołania takiego zapytania możemy wykorzystać przeciążenia metod:

- `execute` – wykonywanie zapytań typu `SELECT`, `INSERT`, `UPDATE`, `DELETE`
- `executeUpdate` – wykonywanie zapytań typu `INSERT`, `UPDATE`, `DELETE`
- `executeQuery` – wykonywanie zapytań typu `SELECT`

Method	DELETE	INSERT	SELECT	UPDATE
<code>stmt.execute()</code>	Yes	Yes	Yes	Yes
<code>stmt.executeQuery()</code>	No	No	Yes	No
<code>stmt.executeUpdate()</code>	Yes	Yes	No	Yes

Method	Return Type	What Is Returned for SELECT	What Is Returned for DELETE/INSERT/UPDATE
<code>stmt.execute()</code>	boolean	true	false
<code>stmt.executeQuery()</code>	ResultSet	The rows and columns returned	n/a
<code>stmt.executeUpdate()</code>	int	n/a	Number of rows added/changed/removed

STATEMENT

```
11: Statement stmt = conn.createStatement();
12: int result = stmt.executeUpdate(
13:     "insert into species values(10, 'Deer', 3)");
14: System.out.println(result); // 1
15: result = stmt.executeUpdate(
16:     "update species set name = '' where name = 'None'");
17: System.out.println(result); // 0
18: result = stmt.executeUpdate(
19:     "delete from species where id = 10");
20: System.out.println(result); // 1
```

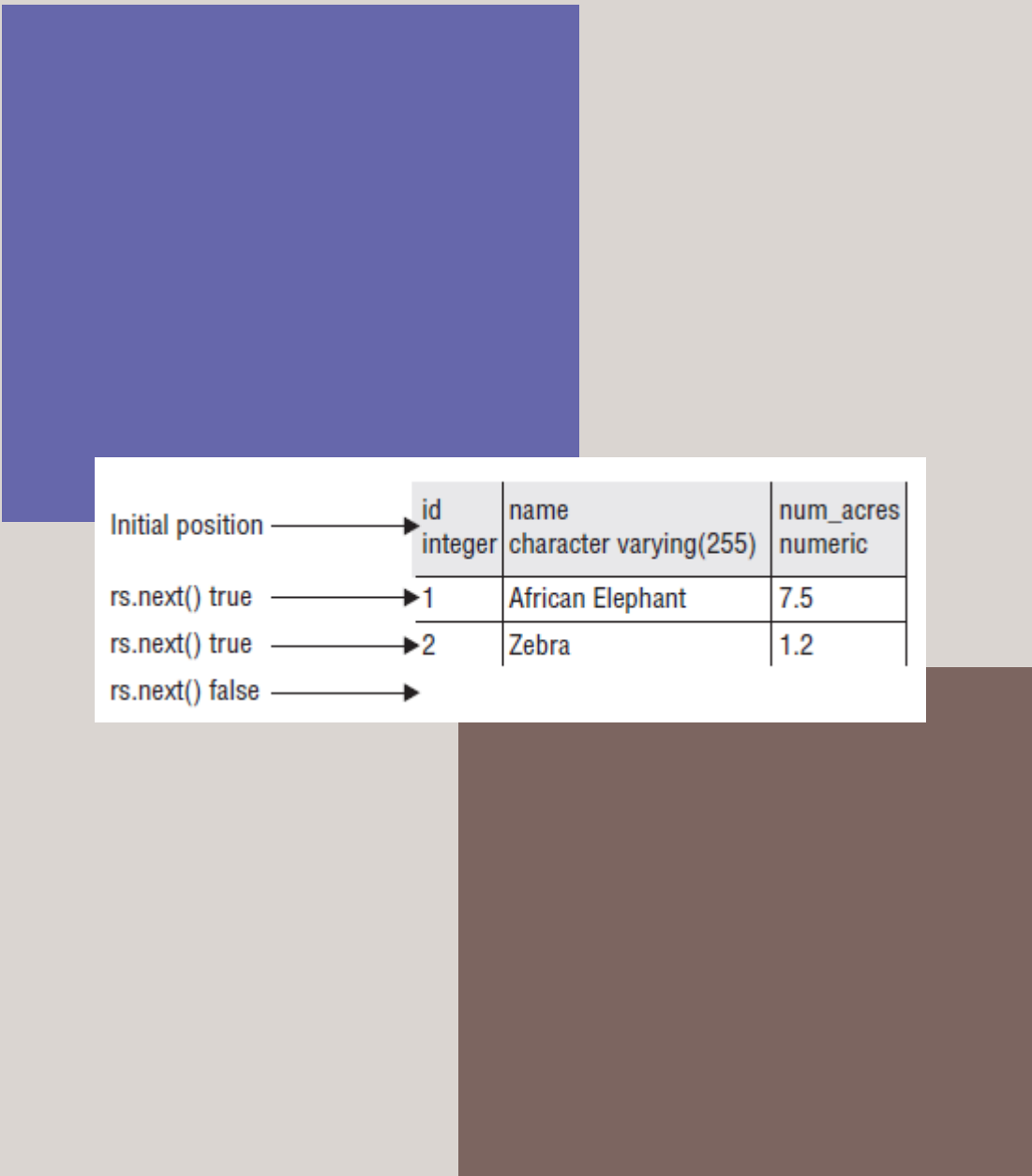
```
ResultSet rs = stmt.executeQuery("select * from species");
```

```
boolean isResultSet = stmt.execute(sql);
if (isResultSet) {
    ResultSet rs = stmt.getResultSet();
    System.out.println("ran a query");
} else {
    int result = stmt.getUpdateCount();
    System.out.println("ran an update");
}
```

```
Connection conn = DriverManager.getConnection("jdbc:derby:zoo");
Statement stmt = conn.createStatement();
int result = stmt.executeUpdate("select * from animal");
```

RESULTSET

Kolejnym ważnym obiektem, który reprezentuje tabelę będącą **wynikiem wykonanego zapytania** jest instancja **ResultSet** (który również **implementuje interfejs AutoCloseable** ale jest on automatycznie zamykany **podczas zamykania instancji Statement**. Opiera się on na kursorze, który możemy przesuwać po tabeli wynikowej, np. do kolejnego wiersza możemy dostać się za pomocą wywołania metody **next()**. Wartość kolumny możemy pobrać za pomocą odpowiedniej metody **getX**, gdzie **X** jest **typem kolumny** (np. String, Boolean czy Byte). Co ważne, **pierwsza kolumna ma indeks 1**.



	id integer	name character varying(255)	num_acres numeric
Initial position	→ 1	African Elephant	7.5
rs.next() true	→ 2	Zebra	1.2
rs.next() true	→		
rs.next() false	→		

RESULTSET

```
20: Map<Integer, String> idToNameMap = new HashMap<>();
21: ResultSet rs = stmt.executeQuery("select id, name from species");
22: while(rs.next()) {
23:     int id = rs.getInt("id");
24:     String name = rs.getString("name");
25:     idToNameMap.put(id, name);
26: }
27: System.out.println(idToNameMap); // {1=African Elephant, 2=Zebra}
```

```
20: Map<Integer, String> idToNameMap = new HashMap<>();
21: ResultSet rs = stmt.executeQuery("select id, name from species");
22: while(rs.next()) {
23:     int id = rs.getInt(1);
24:     String name = rs.getString(2);
25:     idToNameMap.put(id, name);
26: }
27: System.out.println(idToNameMap); // {1=African Elephant, 2=Zebra}
```

PREPAREDSTAMENT

PreparedStatement jest **reprezentacją szkieletu zapytania SQL**. Posiada ona tzw. **placeholdery** tzn. miejsca, które można zastąpić dowolną wartością. W zapytaniu reprezentowane są poprzez znak **?**. Wartość takiego placeholdera możemy zastąpić **odpowiednią metodą setX**, w zależności od używanego typu danych, np. `setInt` czy `setString`. Podczas wykorzystywania tych metod, musimy podać **indeks** takiego **placeholdera** (**pierwszy z nich ma indeks 1**).

```
public static void main(String[] args) throws SQLException {
    try (Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/1z0809", "root", "my-secret-pw");
        PreparedStatement statement = connection.prepareStatement(
            "SELECT * FROM Persons WHERE FirstName = ? OR PersonId = ?")) {

        statement.setString(1, "Andrzej");
        statement.setInt(2, 1);
        final ResultSet resultSet = statement.executeQuery();
        while (resultSet.next()) {
            System.out.println("First name is " + resultSet.getString(2));
        }
    }
}
```

CALLABLESTATEMENT

CallableStatement jest kolejnym typem obiektu Statement, który **służy do wykonywania procedur**. CallableStatement, podobnie jak PreparedStatement, **pozwalą na korzystanie placeholderów**.

```
public void calculateProductStatistics(
    Connection connection, String productName) throws SQLException {
    CallableStatement calculateStatisticsStatement = connection.prepareCall(
        "{call calculateStatistics(?)}");
    calculateStatisticsStatement.setString(1, productName);
    ResultSet result = calculateStatisticsStatement.executeQuery();
}
```

LOKALIZCJA

Lokalizacja to specyficzne dla danego języka, kraju i regionu reguły dotyczące prezentacji różnych informacji (formatowania liczb, formatowania dat, pisowni tekstów, porządku alfabetycznego, itp).

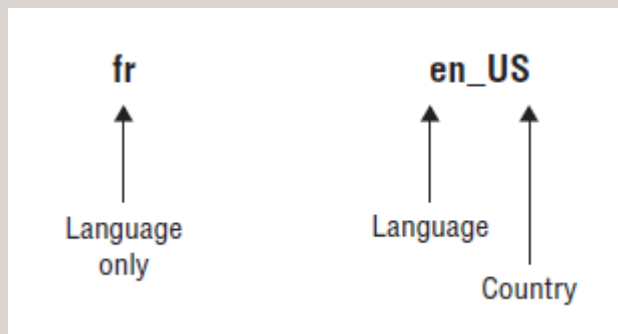
LOCALE

Klasa **Locale** (zdefiniowana w pakiecie **java.util**).

W celu pobrania bieżącej lokalizacji, możemy posłużyć się następującym fragmentem kodu:

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

Format wyświetlania lokalizacji jest ściśle określony!



```
US      // can have a language without a country, but not the reverse  
enUS    // missing underscore  
US_en   // the country and language are reversed  
EN      // language must be lowercase
```

LOCALE - KREACJA

Istnieją trzy sposoby kreacji obiektu **Locale**:

- wykorzystanie stałych

```
System.out.println(Locale.GERMAN); // de
System.out.println(Locale.GERMANY); // de_DE
```

- wykorzystanie konstruktora:

```
System.out.println(new Locale("fr")); // fr
System.out.println(new Locale("hi", "IN")); // hi_IN
```

- wykorzystanie budowniczego:

```
Locale l1 = new Locale.Builder()
    .setLanguage("en")
    .setRegion("US")
    .build();
Locale l2 = new Locale.Builder()
    .setRegion("US")
    .setLanguage("en")
    .build();
```

```
Locale l2 = new Locale.Builder() // bad but legal
    .setRegion("us")
    .setLanguage("EN")
    .build();
```

RESOURCEBUNDLE

Obiekt **ResourceBundle** (paczka zasobów) jest obiektem przechowującym zestaw danych uzależnionych od lokalizacji. Obiekt ten może być rozumiany jako zestaw wpisów: **klucz – wartość** (mapa). Klucz jednoznacznie identyfikuje obiekt specyficzny dla ustawień regionalnych. Obiekt ResourceBundle może być obsługiwany poprzez odpowiedni **plik tekstowy** lub **obiekt klasy** zawierający właściwe wpisy (pary).



RESOURCEBUNDLE – PLIK *.PROPERTIES

Nazwa pliku składa się z **nazwy zasobu** oraz **członu wskazującego na konkretną lokalizację**.

```
import java.util.*;
public class ZooOpen {

    public static void main(String[] args) {
        Locale us = new Locale("en", "US");
        Locale france = new Locale("fr", "FR");
        printProperties(us);
        System.out.println();
        printProperties(france);
    }

    public static void printProperties(Locale locale) {
        ResourceBundle rb = ResourceBundle.getBundle("Zoo", locale);
        System.out.println(rb.getString("hello"));
        System.out.println(rb.getString("open"));
    }
}
```

Zoo_en.properties

hello=Hello
open=The zoo is open.

Zoo_fr.properties

hello=Bonjour
open=Le zoo est ouvert

RESOURCEBUNDLE

– LISTRESOURCEBUNDLE

W zdecydowanej większości przypadków paczka zasobów w postaci pliku **properties** jest wystarczająca. Głównym ograniczeniem tego rozwiązania jest możliwość deklarowania wartości **wyłącznie w postaci łańcuchów znaków**.

Zalety **ListResourceBundle**:

- Możliwość użycia wartości typu innego niż String
- Możliwość definiowania wartości w trakcie wykonywania aplikacji (runtime)

```
import java.util.*;
public class Zoo_en extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "hello", "Hello" },
            { "open", "The zoo is open" } };
    }
}
```

RESOURCEBUNDLE – LISTRESOURCEBUNDLE

```
package resourcebundles;
import java.util.*;
public class Tax_en_US extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] { { "tax", new UsTaxCode() } };
    }
    public static void main(String[] args) {
        ResourceBundle rb = ResourceBundle.getBundle(
            "resourcebundles.Tax", Locale.US);
        System.out.println(rb.getObject("tax"));
    }
}
```



DZIĘKUJĘ ZA UWAGĘ

