

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

- **Wielowątkowość**

tworzenie wątków z użyciem Runnable oraz Callable; wykonywania zadań z użyciem ExecutorService; identyfikacja problemów: deadlock, starvation, livelock, race conditions; konstrukcja synchronized; klasy z pakietu java.util.concurrent.atomic; kontrola kolejności wykonywania wątków; kolekcje i klasy z pakietu java.util.concurrent (CyclicBarrier, CopyOnWriteArrayList); framework Fork/Join; strumienie równoległe

- **Tworzenie aplikacji z wykorzystaniem JDBC**

interfejsy: Driver, Connection, Statement, ResultSet; użycie elementów JDBC API do nawiązania połączenia; wysyłanie zapytań do bazy oraz odczyt wyników, iteracja po wynikach oraz zamykanie obiektów JDBC API

- **Lokalizacja**

klasa Locale; utworzenie oraz odczyt pliku properties; tworzenie zasobów lokalizacyjnych



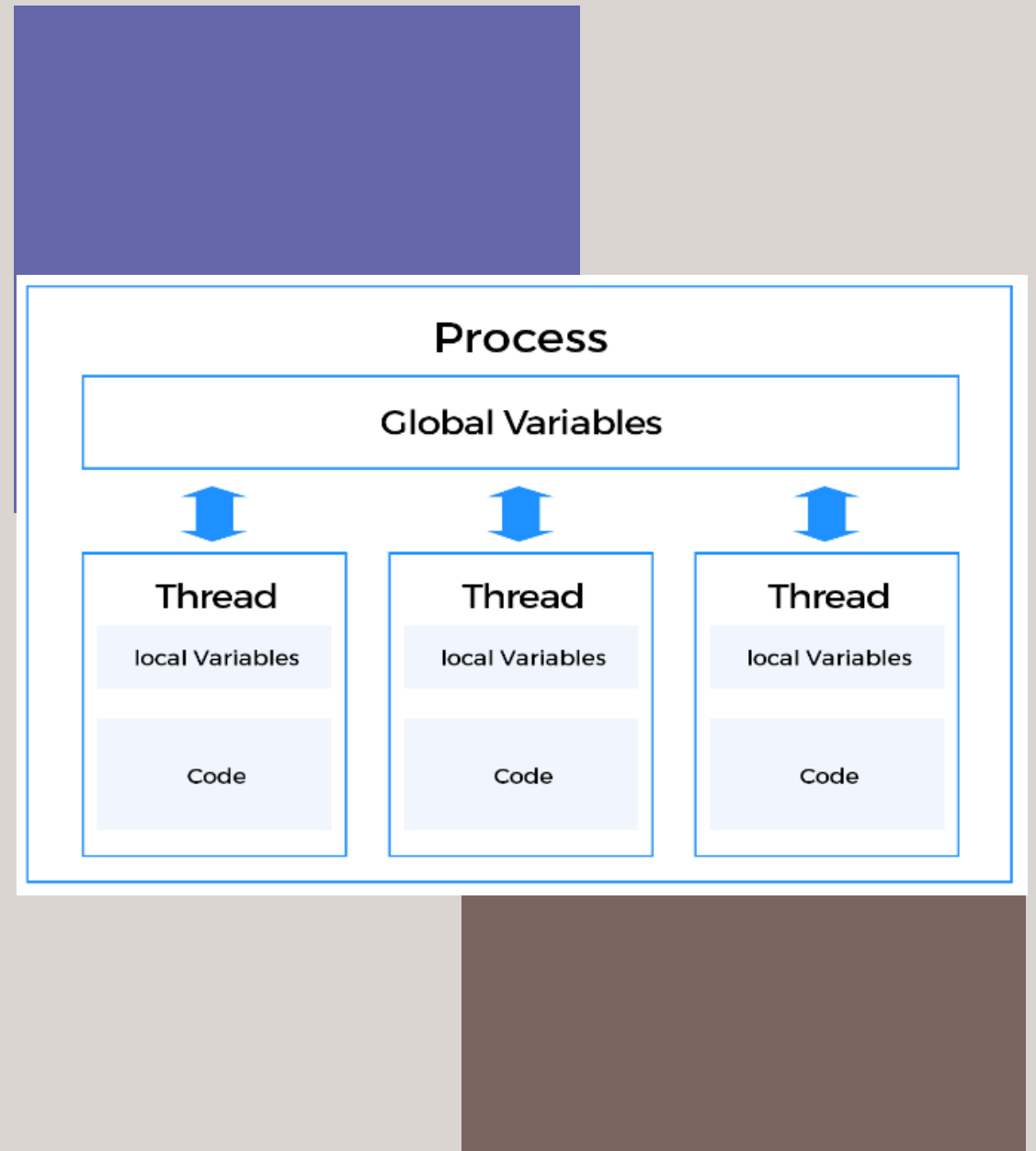


# WIELOWĄTKOWOŚĆ

**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.

# WĄTEK A PROCES

W momencie, gdy uruchamiamy naszą aplikację, na poziomie systemu operacyjnego **uruchamiamy nowy proces**. Listę takich procesów możemy uzyskać za pomocą komendy **ps**. W ramach jednego procesu może istnieć wiele wątków. Wątki **posiadają wspólną przestrzeń adresową** oraz **otwarte struktury systemowe** (jak np. **otwarte pliki**), procesy z kolei **posiadają niezależne przestrzenie adresowe**.



# THREAD

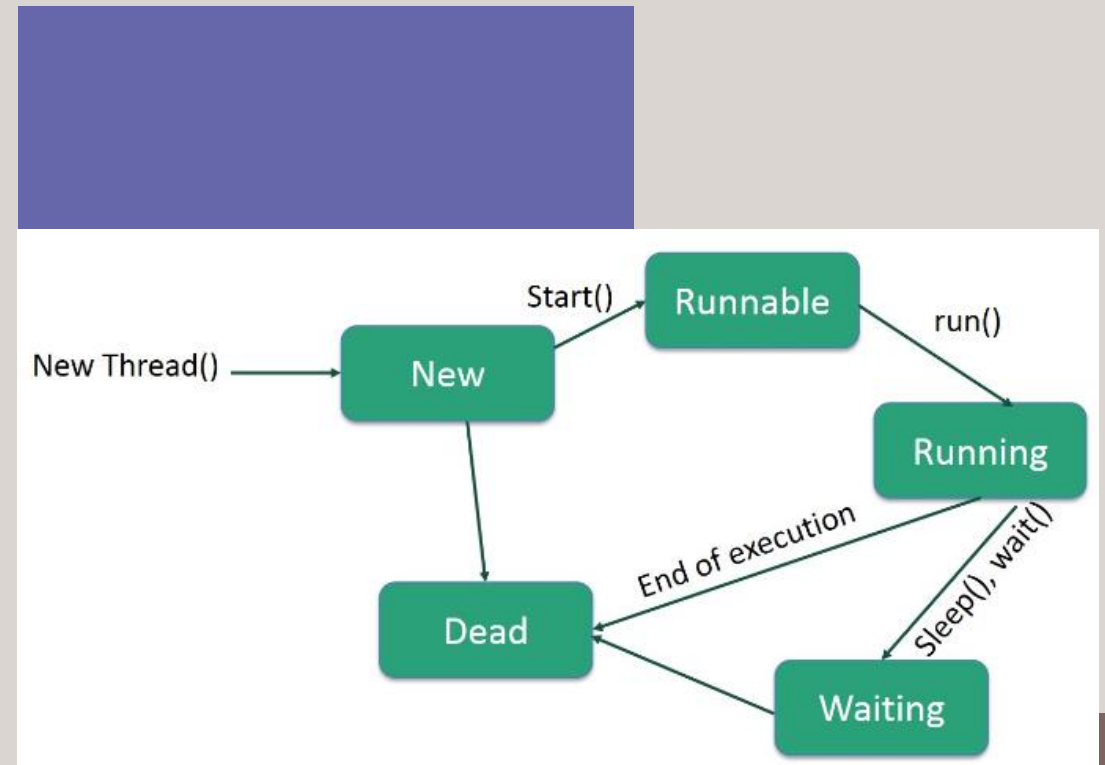
**Thread** jest **wątkiem wykonania w programie**.

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");
    }
}
```



# PRZERWANIE WĄTKU

W zależności od okoliczności i stanu aplikacji, możemy czasem **chcieć przerwać pracę wykonywaną przez pewien wątek**. Nie da się jednak w trywialny sposób przerwać wątku. Możemy, co najwyżej, **wysłać prośbę o zatrzymanie takiego wątku**. To programista, w kodzie osobnego wątku, **decyduje co z tym faktem zrobić**.

```
public class ThreadsExample {  
    public static void main(String[] args) {  
        final Thread sleepingThread = new Thread(new SleepingThread());  
        sleepingThread.start();  
        sleepingThread.interrupt(); // wysłanie prośby o zatrzymanie  
    }  
}  
  
class SleepingThread implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("I will go to sleep");  
        try {  
            Thread.sleep(3000L);  
        } catch (InterruptedException e) {  
            System.out.println("I was interrupted during sleep");  
        }  
        System.out.println("I am exiting");  
    }  
}
```

# PRZERWANIE WĄTKU

```
public class ThreadsExample {
    public static void main(String[] args) {
        final Thread sleepingThread = new Thread(new SleepingThread());
        sleepingThread.start();
        sleepingThread.interrupt();
    }
}

class SleepingThread implements Runnable {
    @Override
    public void run() {
        final List<Integer> ints = new ArrayList<>();
        for (int idx = 0; idx < 1000; idx++) {
            ints.add(new Random().nextInt());
        }
        if (Thread.currentThread().isInterrupted()) {
            return;
        }
        final int sum = ints.stream().mapToInt(value -> value).sum();
        System.out.println("Sum is " + sum);
    }
}
```

# 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++;
    }

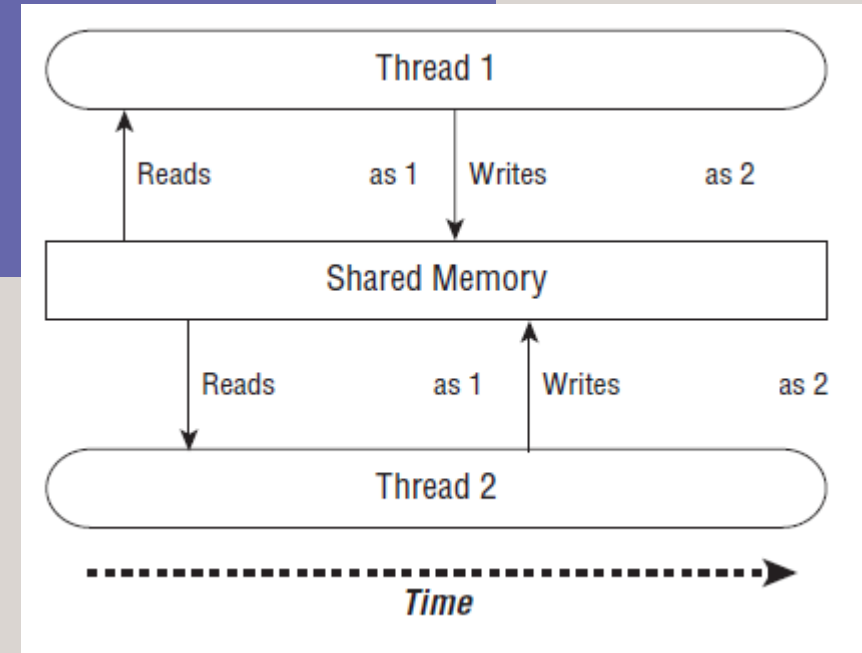
    public Integer getLeft() {
        return left;
    }

    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();
    }
}
```

# RACE CONDITION

To co udało Ci zaobserwować w poprzednim przykładzie to tak zwany **wyścig** (ang. **race condition**). Taka sytuacja zachodzi jeśli **kilka wątków jednocześnie modyfikuje zmienną**, która do takiej równoległej zmiany nie jest przystosowana. Tylko dlatego wartości atrybutów left oraz right miały różne wartości? Działo się tak dlatego, że **operacja inkrementacji nie jest operacją atomową**.



# 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");  
}
```

# JOIN

Dodatkowe wątki w aplikacjach często wykorzystujemy do wyliczenia pewnych danych, które następnie przetwarzamy, np. w głównym wątku. Zanim będziemy mogli rozpocząć przetwarzanie, **jesteśmy zmuszeni poczekać na zakończenie wszystkich wątków wyliczających dane**. Aby poczekać na zakończenie wątku, musimy wykorzystać metodę **join**. Dostępne są przeciążenia:

- **bezargumentowe, czekające tak długo, aż wątek się zakończy**
- wersje z argumentami, gdzie możemy podać ilość milisekund, **oznaczających maksymalny czas czekania na zakończenie wątku**.

```
public class ThreadsExample {
    public static void main(String[] args) throws InterruptedException {
        final List<Integer> ints = new ArrayList<>();
        final Thread threadA = new Thread(new SimpleThread(ints));
        final Thread threadB = new Thread(new SimpleThread(ints));

        threadA.start();
        threadB.start();

        threadA.join(1000L);
        threadB.join(1000L);
        System.out.println(ints.size());
    }
}

class SimpleThread implements Runnable {

    private final List<Integer> ints;

    SimpleThread(final List<Integer> ints) {
        this.ints = ints;
    }

    @Override
    public void run() {
        synchronized (this.ints) {
            ints.add(new Random().nextInt());
        }
    }
}
```

# 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";  
    }  
}
```

# FUTURE

**Future** to **interfejs reprezentujący przyszły wynik działania metody asynchronicznej**, który ostatecznie zostanie zwrócony w przyszłości po zakończeniu przetwarzania operacji. Wartość działania operacji jest możliwa do pobrania za pomocą metody **get()**, która działa **podobnie jak metoda join w klasie Thread**, tzn. blokuje aktualny wątek i czeka na moment, aż oczekiwany wynik będzie dostępny.



# FUTURE

`boolean isDone()`

Returns true if the task was completed, threw an exception, or was cancelled.

`boolean isCancelled()`

Returns true if the task was cancelled before it completely normally.

`boolean cancel()`

Attempts to cancel execution of the task.

`V get()`

Retrieves the result of a task, waiting endlessly if it is not yet available.

`V get(long timeout,  
TimeUnit unit)`

Retrieves the result of a task, waiting the specified amount of time. If the result is not ready by the time the timeout is reached, a checked `TimeoutException` will be thrown.

# 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

```
void execute(Runnable command)
```

Executes a Runnable task at some point in the future

```
Future<?> submit(Runnable task)
```

Executes a Runnable task at some point in the future and returns a Future representing the task

```
<T> Future<T> submit(Callable<T> task)
```

Executes a Callable task at some point in the future and returns a Future representing the pending results of the task

```
<T> List<Future<T>> invokeAll(  
Collection<? extends Callable<T>> tasks)  
throws InterruptedException
```

Executes the given tasks, synchronously returning the results of all tasks as a Collection of Future objects, in the same order they were in the original collection

```
<T> T invokeAny(  
Collection<? extends Callable<T>> tasks)  
throws InterruptedException,  
ExecutionException
```

Executes the given tasks, synchronously returning the result of one of finished tasks, cancelling any unfinished tasks

# 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();
    }
}
```

# WYKONYWANIE ZADAŃ

- **invokeAny()** - **ExecutorService** w swojej puli wątków zaczyna wykonywać listę wejściowych zadań. Zwraca rezultat rozpoczętych zadań, które zostały zakończone sukcesem w momencie, gdy pierwszy z nich zakończył swoje działanie. Pozostałe, niezakończone zadania, zostaną anulowane.

```
public class HomeTasks {  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newFixedThreadPool(2);  
        List<Callable<String>> tasks = Arrays.asList(  
            () -> {  
                System.out.println("Wątek: " + Thread.currentThread().getName());  
                System.out.println("Robię zakupy");  
                Thread.sleep(5000);  
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Zrobiono zakupy!");  
                return "Zrobiono zakupy!";  
            },  
            () -> {  
                System.out.println("Wątek: " + Thread.currentThread().getName());  
                System.out.println("Mycie naczyń");  
                Thread.sleep(2000);  
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Umyto naczynia");  
                return "umyto naczynia";  
            },  
            () -> {  
                System.out.println("Wątek: " + Thread.currentThread().getName());  
                System.out.println("Sprzątanie pokoju");  
                Thread.sleep(1000);  
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Posprzątano pokój");  
                return "posprzątano pokój";  
            }  
        );  
        try {  
            String firstResult = executorService.invokeAny(tasks);  
            System.out.println("PIERWSZY WYNIK: " + firstResult);  
        } catch (InterruptedException | ExecutionException e) {  
            e.printStackTrace();  
        }  
        executorService.shutdown();  
    }  
}
```

# WYKONYWANIE ZADAŃ

- **invokeAll** - wykonuje **wszystkie zadania** typu Callable i zwraca listę rezultatów typu List<Future<T>>.

```
public class HomeTasks {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        List<Callable<String>> tasks = Arrays.asList(
            () -> {
                System.out.println("Wątek: " + Thread.currentThread().getName());
                System.out.println("Robię zakupy");
                Thread.sleep(5000);
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Zrobiono zakupy!");
                return "Zrobiono zakupy!";
            },
            () -> {
                System.out.println("Wątek: " + Thread.currentThread().getName());
                System.out.println("Mycie naczyń");
                Thread.sleep(2000);
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Umyto naczynia");
                return "umyto naczynia";
            },
            () -> {
                System.out.println("Wątek: " + Thread.currentThread().getName());
                System.out.println("Sprzątanie pokoju");
                Thread.sleep(1000);
                System.out.println("Wątek: " + Thread.currentThread().getName() + ". Posprzątano pokój");
                return "posprzątano pokój";
            }
        );
        try {
            List<Future<String>> futures = executorService.invokeAll(tasks);
            for (Future<String> future : futures) {
                System.out.println(future.get());
            }
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        executorService.shutdown();
    }
}
```

# PLANOWANIE ZADAŃ

Ta implementacja **ScheduledExecutorService** umożliwia zaplanowanie wykonywania operacji **po określonym czasie lub z pewnym interwałem**. W ramach metod tej implementacji ExecutorService możemy wyróżnić poniższe metody:

- **scheduleAtFixedRate**
- **scheduleWithFixedDelay**





# PLANOWANIE ZADAŃ

## - SCHEDULEATFIXEDRATE

```
public class SchedulerExecutorDemo {
    public static void main(String[] args) throws InterruptedException {
        DateFormat df = new SimpleDateFormat("dd:MM:yy:HH:mm:ss");
        ScheduledExecutorService executorService = Executors.newScheduledThreadPool(1);
        executorService.scheduleAtFixedRate(() -> {
            System.out.println("Start coffee!: " + df.format(Calendar.getInstance().getTime()));
            try {
                Thread.sleep(5000);
                System.out.println("finish coffee!: " + df.format(Calendar.getInstance().getTime()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 1, 6, TimeUnit.SECONDS);
        Thread.sleep(15000L);
        executorService.shutdown();
    }
}
```

# PLANOWANIE ZADAŃ -SCHEDULEWITHFIXEDDELAY

```
public class ScheduledWithFixedDelayDemo {
    public static void main(String[] args) throws InterruptedException {
        DateFormat df = new SimpleDateFormat("dd:MM:yy:HH:mm:ss");
        ScheduledExecutorService executorService = Executors.newScheduledThreadPool(1);
        executorService.scheduleWithFixedDelay(() -> {
            System.out.println("Start coffee!: " + df.format(Calendar.getInstance().getTime()));
            try {
                Thread.sleep(5000);
                System.out.println("finish coffee!: " + df.format(Calendar.getInstance().getTime()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 1, 6, TimeUnit.SECONDS);
        Thread.sleep(15000L);
        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());
    }
}
```

# KOLEKCJE WSPÓŁBIEŻNE

```
public class ZooManager {  
    private Map<String,Object> foodData = new HashMap<String,Object>();  
    public synchronized void put(String key, String value) {  
        foodData.put(key, value);  
    }  
    public synchronized Object get(String key) {  
        return foodData.get(key);  
    }  
}
```

```
public class ZooManager {  
    private Map<String,Object> foodData = new ConcurrentHashMap<String,Object>();  
    public void put(String key, String value) {  
        foodData.put(key, value);  
    }  
    public Object get(String key) {  
        return foodData.get(key);  
    }  
}
```

# KOLEKCJE WSPÓŁBIEŻNE

```
Map<String, Object> foodData = new HashMap<String, Object>();  
foodData.put("penguin", 1);  
foodData.put("flamingo", 2);  
for(String key: foodData.keySet())  
    foodData.remove(key);
```

```
Map<String, Object> foodData = new ConcurrentHashMap<String, Object>();  
foodData.put("penguin", 1);  
foodData.put("flamingo", 2);  
for(String key: foodData.keySet())  
    foodData.remove(key);
```

# KLASY KOLEKCJI WSPÓŁBIEŻNYCH

Class Name	Java Collections Framework Interface	Elements Ordered?	Sorted?	Blocking?
ConcurrentHashMap	ConcurrentMap	No	No	No
ConcurrentLinkedDeque	Deque	Yes	No	No
ConcurrentLinkedQueue	Queue	Yes	No	No
ConcurrentSkipListMap	ConcurrentMap SortedMap NavigableMap	Yes	Yes	No
ConcurrentSkipListSet	SortedSet NavigableSet	Yes	Yes	No
CopyOnWriteArrayList	List	Yes	No	No
CopyOnWriteArraySet	Set	No	No	No
LinkedBlockingDeque	BlockingQueue BlockingDeque	Yes	No	Yes
LinkedBlockingQueue	BlockingQueue	Yes	No	Yes

# COPYONWRITEARRAYLIST

**CopyOnWriteArrayList** to ulepszona implementacja **ArrayList**, w której operacje takie jak: **add(...)**, **set(...)**, **remove(...)**, **powodują utworzenie świeżej kopii wewnętrznej tablicy** przechowującej elementy. Lista ta jest **przeznaczona do zastosowań wielowątkowych**.

Wszystkie operacje modyfikujące jej stan są bardzo kosztowne, dlatego najlepiej jest **używać jej tylko w sytuacjach, gdzie potrzebujemy częstych odczytów i sporadycznych modyfikacji listy**.

```
List<Integer> list = new CopyOnWriteArrayList<>(Arrays.asList(4,3,52));
for(Integer item: list) {
    System.out.print(item+" ");
    list.add(9);
}
System.out.println();
System.out.println("Size: "+list.size());
```

# CYCLICBARRIER

Bariera cykliczna jest konstrukcją synchronizującą **umożliwiającą oczekiwanie przez poszczególne wątki na osiągnięcie określonego punktu zwanego barierą.**

Bariera ta nazywa się barierą cykliczną ze względu na późniejszą **możliwość re-użycia** (po zwolnieniu wszystkich wątków oczekujących na tej blokadzie).





# CYCLICBARRIER

```
import java.util.concurrent.*;

public class LionPenManager {

    private void removeAnimals() { System.out.println("Removing animals"); }
    private void cleanPen() { System.out.println("Cleaning the pen"); }
    private void addAnimals() { System.out.println("Adding animals"); }

    public void performTask() {
        removeAnimals();
        cleanPen();
        addAnimals();
    }

    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newFixedThreadPool(4);
            LionPenManager manager = new LionPenManager();
            for(int i=0; i<4; i++)
                service.submit(() -> manager.performTask());
        } finally {
            if(service != null) service.shutdown();
        }
    }
}
```

```
public class LionPenManager {

    private void removeAnimals() { System.out.println("Removing animals"); }
    private void cleanPen() { System.out.println("Cleaning the pen"); }
    private void addAnimals() { System.out.println("Adding animals"); }

    public void performTask(CyclicBarrier c1, CyclicBarrier c2) {
        try {
            removeAnimals();
            c1.await();
            cleanPen();
            c2.await();
            addAnimals();
        } catch (InterruptedException | BrokenBarrierException e) {
            // Handle checked exceptions here
        }
    }

    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newFixedThreadPool(4);

            LionPenManager manager = new LionPenManager();
            CyclicBarrier c1 = new CyclicBarrier(4);
            CyclicBarrier c2 = new CyclicBarrier(4,
                () -> System.out.println("*** Pen Cleaned!"));
            for(int i=0; i<4; i++)
                service.submit(() -> manager.performTask(c1,c2));
        } finally {
            if(service != null) service.shutdown();
        }
    }
}
```

# 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();
```



# PRZETWARZANIE STRUMIENI RÓWNOLEGŁYCH

```
Arrays.asList(1,2,3,4,5,6)
    .parallelStream()
    .forEach(s -> System.out.print(s+" "));
```

```
Arrays.asList(1,2,3,4,5,6)
    .parallelStream()
    .forEachOrdered(s -> System.out.print(s+" "));
```

```
Arrays.asList("jackal","kangaroo","lemur")
    .parallelStream()
    .map(s -> {System.out.println(s); return s.toUpperCase();})
    .forEach(System.out::println);
```

```
List<Integer> data = Collections.synchronizedList(new ArrayList<>());
Arrays.asList(1,2,3,4,5,6).parallelStream()
    .map(i -> {data.add(i); return i;}) // AVOID STATEFUL LAMBDA EXPRESSIONS!
    .forEachOrdered(i -> System.out.print(i+" "));

System.out.println();
for(Integer e: data) {
    System.out.print(e+" ");
}
```



# STRUMIENIE RÓWNOLEGŁE

## - REDUKCJE

- The *identity* must be defined such that for all elements in the stream  $u$ , `combiner.apply(identity, u)` is equal to  $u$ .
- The *accumulator* operator  $op$  must be associative and stateless such that  $(a \ op \ b) \ op \ c$  is equal to  $a \ op \ (b \ op \ c)$ .
- The *combiner* operator must also be associative and stateless and compatible with the identity, such that for all  $u$  and  $t$  `combiner.apply(u, accumulator.apply(identity, t))` is equal to `accumulator.apply(u, t)`.

```
System.out.print(Arrays.asList(1,2,3,4,5,6).stream().findAny().get());
```

```
System.out.println(Arrays.asList(1,2,3,4,5,6)  
    .parallelStream()  
    .reduce(0, (a,b) -> (a-b))); // NOT AN ASSOCIATIVE ACCUMULATOR
```

```
System.out.println(Arrays.asList("w","o","l","f")  
    .parallelStream()  
    .reduce("X", String::concat));
```

# STRUMIENIE RÓWNOLEGŁE - KOLEKTORY

```
Stream<String> stream = Stream.of("w", "o", "l", "f").parallel();  
Set<String> set = stream.collect(Collectors.toSet());  
System.out.println(set); // [f, w, l, o]
```

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears").parallel();  
ConcurrentMap<Integer, String> map = ohMy  
    .collect(Collectors.toConcurrentMap(String::length, k -> k,  
        (s1, s2) -> s1 + "," + s2));  
System.out.println(map); // {5=lions,bears, 6=tigers}  
System.out.println(map.getClass()); // java.util.concurrent.ConcurrentHashMap
```

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears").parallel();  
ConcurrentMap<Integer, List<String>> map = ohMy.collect(  
    Collectors.groupingByConcurrent(String::length);  
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

# 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.

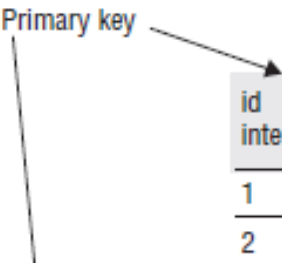
Główne funkcjonalności, jakie JDBC realizuje to:

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

# STRUKTURA TABEL BAZODANOWYCH

Podstawowe pojęcia / zagadnienia:

- **Tabela** – struktura składająca się z określonej liczby kolumn oraz wierszy
- **Klucz główny (ang. primary key)** - jednoznacznie definiuje każdy rekord danej tabeli (pojedynczy atrybut lub zestaw atrybutów) – w sposób unikalny



column			row
id	name	num_acres	
integer	character varying(255)	numeric	
1	African Elephant	7.5	
2	Zebra	1.2	

id	species_id	name	date_born
integer	integer	character varying(255)	timestamp without time zone
1	1	Elsa	2001-05-06 02:15:00
2	2	Zelda	2002-08-15 09:12:00
3	1	Ester	2002-09-09 10:36:00
4	1	Eddie	2010-06-08 01:24:00
5	2	Zoe	2005-11-12 03:44:00

# PODSTAWOWE ZAPYTANIA SQL

Podstawowe zapytania SQL:

- **INSERT** – dodanie nowego rekordu do tabeli

```
INSERT INTO species VALUES (3, 'Asian Elephant', 7.5);
```

- **SELECT** – wyszukiwanie danych w tabeli

```
SELECT * FROM SPECIES WHERE ID = 3;
```

```
SELECT NAME, NUM_ACRES FROM SPECIES WHERE ID = 3;
```

```
SELECT COUNT(*), SUM(num_acres) FROM SPECIES;
```

- **UPDATE** – modyfikacja danych w tabeli

```
UPDATE SPECIES SET NUM_ACRES = NUM_ACRES + .5 WHERE NAME = 'Asian Elephant';
```

- **DELETE** – usuwanie rekordów z tabeli

```
DELETE FROM SPECIES WHERE NAME = 'Asian Elephant';
```



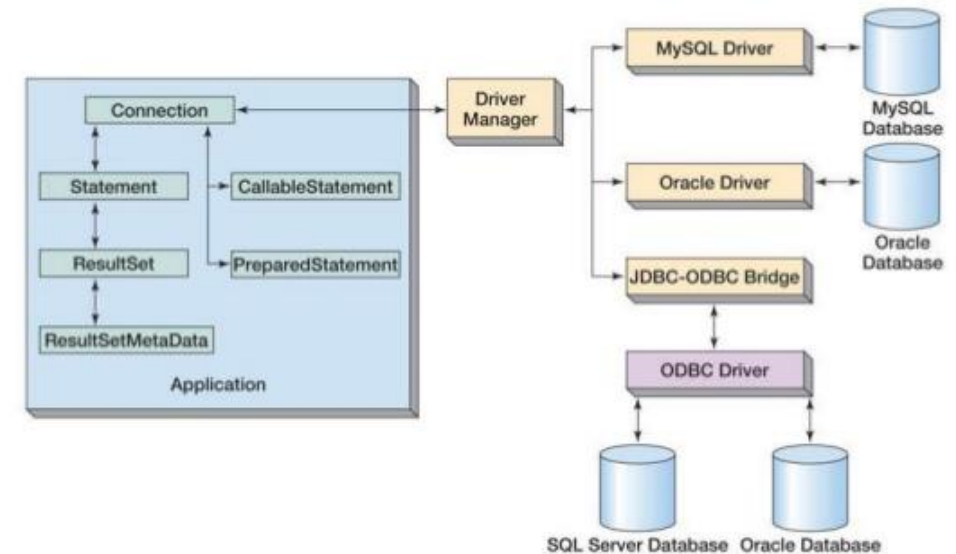


# KOMPONENTY

JDBC składa się z czterech komponentów:

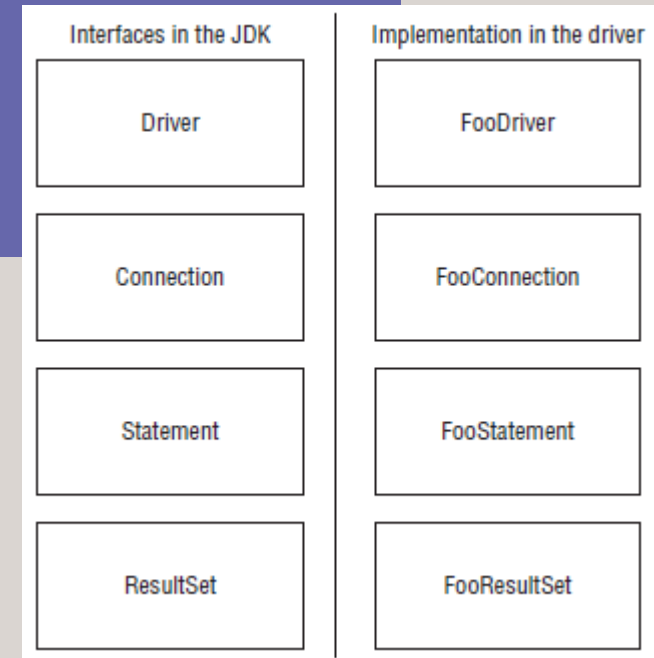
- **JDBC API** - jest to najważniejsza część JDBC, która pozwala, za pomocą wywoływania metod z poziomu kodu, wywoływać zapytania na bazie danych i pobierać ich wyniki. Składniki API znajdziemy w pakietach `java.sql` i `javax.sql`.
- **JDBC Driver Manager** - definiuje w jaki sposób nawiązać połączenie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



# KOMPONENTY JDBC

```
package com.wiley.ocp.connection;

import java.sql.*;

public class MyFirstDatabaseConnection {

    public static void main(String[] args) throws SQLException {
        String url = "jdbc:derby:zoo";
        try (Connection conn = DriverManager.getConnection(url);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("select name from animal")) {

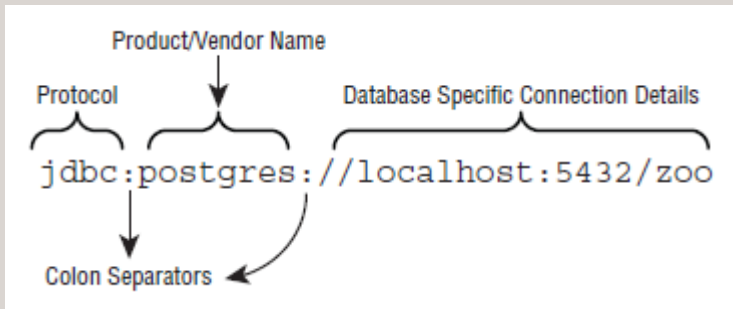
            while (rs.next())
                System.out.println(rs.getString(1));

        }
    }
}
```

# 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

W celu wykonania jakiejkolwiek interakcji z bazą danych musimy najpierw się z nią połączyć. **Każdy serwer bazodanowy** (np. MySQL, MariaDB, Oracle DB czy Microsoft SQL Server) **posiada nieco inną implementację** i w celu nawiązania połączenia do każdego z nich za pomocą takiej samej metody, **potrzebna jest warstwa łącząca**. Taką warstwą jest **Driver**, czyli sterownik, który jest **specyficzny dla konkretnej implementacji bazy danych**.



# 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**



# NAWIAZYWANIE POŁĄCZENIA Z BAZĄ DANYCH

W sytuacji, w której Java nie odnajdzie w przestrzeni klas właściwej implementacji sterownika umożliwiającego nawiązanie połączenia z bazą danych, otrzymujemy wyjątek.

```
import java.sql.*;
public class TestConnect {

    public static void main(String[] args) throws SQLException {
        Connection conn = DriverManager.getConnection("jdbc:derby:zoo");
        System.out.println(conn);
    }
}
```

```
Exception in thread "main" java.sql.SQLException: No suitable driver found for
jdbc:derby:zoo
    at java.sql.DriverManager.getConnection(DriverManager.java:689)
    at java.sql.DriverManager.getConnection(DriverManager.java:270)
```

# 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);
```



# RESULTSETTYPE

Domyślnie, **ResultSet** obsługuje tryb **TYPE\_FORWARD\_ONLY**. Umożliwia on poruszanie się po zbiorze wyniku tylko raz, wyłącznie w jednym kierunku. Pozostałe dwa tryby, tj. **TYPE\_SCROLL\_INSENSITIVE** oraz **TYPE\_SCROLL\_SENSITIVE** umożliwiają przemieszczanie się po zbiorze wyników w dowolnym kierunku. Dodatkowo, tryb **TYPE\_SCROLL\_SENSITIVE** uwzględnia podczas przeszukiwania zbioru zmiany, które miały miejsce w tym samym czasie na bazie danych. Tryb ten nie jest wspierany przez wszystkie bazy danych.





ResultSet Type	Can Go Backward	See Latest Data from Database Table	Supported by Most Drivers
ResultSet.TYPE_FORWARD_ONLY	No	No	Yes
ResultSet.TYPE_SCROLL_INSENSITIVE	Yes	No	Yes
ResultSet.TYPE_SCROLL_SENSITIVE	Yes	Yes	No



# CONCURRENCY MODE

Domyślnie, ResultSet obsługuje tryb **CONCUR\_READ\_ONLY**. Tryb ten oznacza, iż nie ma możliwości wykonania na danym zbiorze jakichkolwiek modyfikacji. W zdecydowanej większości przypadku, w ramach modyfikacji zawartości struktur bazodanowych posługujemy się dedykowanymi operacjami: **INSERT**, **UPDATE**, **DELETE**. Trybem umożliwiającym modyfikację zbioru wynikowego jest **CONCUR\_UPDATABLE**. Tryb ten nie jest wspierany przez wszystkie bazy danych.



ResultSet Type	Can Read Data	Can Update Data	Supported by All Drivers
ResultSet.CONCUR_READ_ONLY	Yes	Yes	No
ResultSet.CONCUR_UPDATABLE	Yes	No	Yes

# 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

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
```

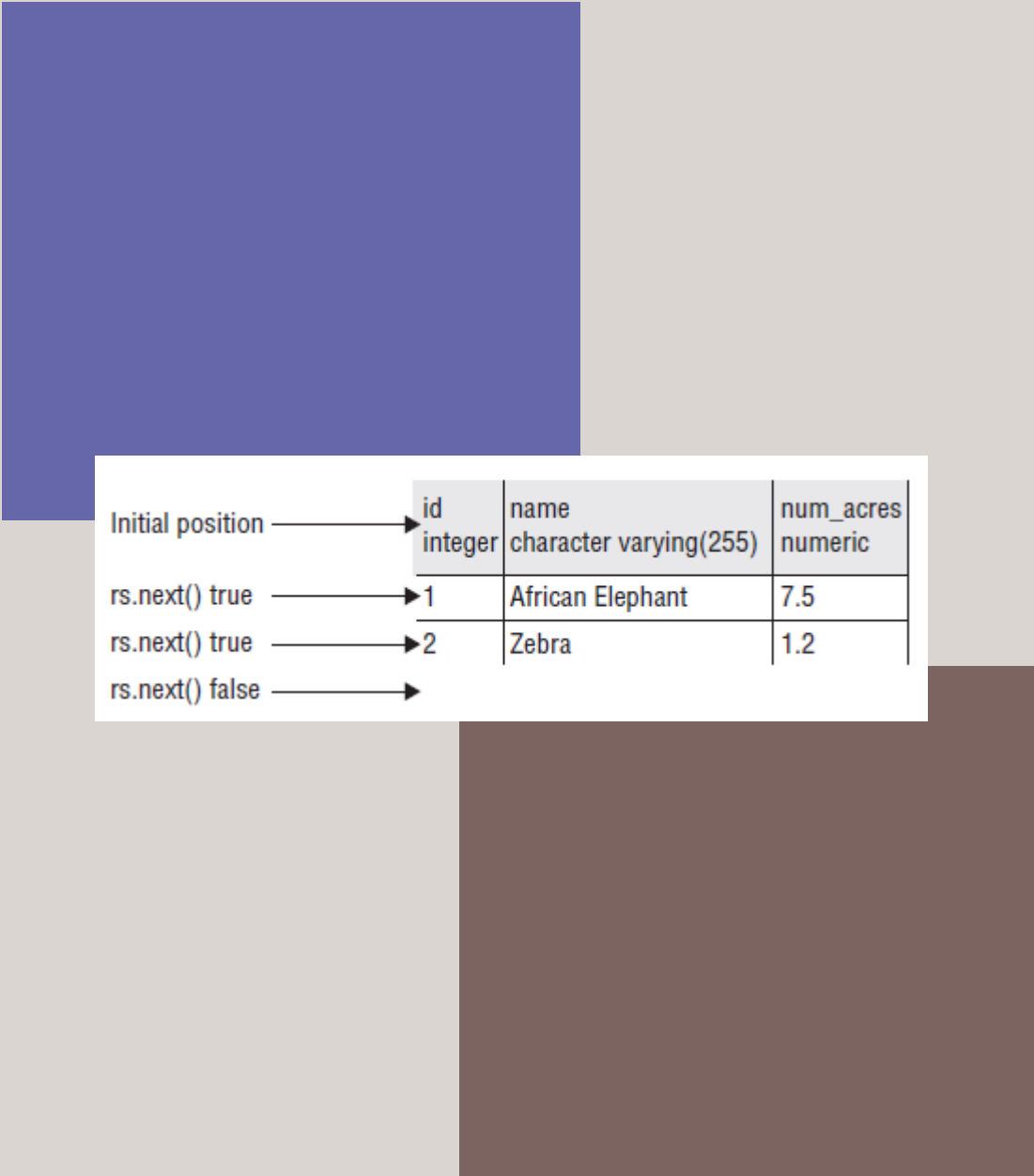
```
ResultSet rs = stmt.executeQuery("select * from species");
```

```
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}
```



# RESULTSET - BŁĘDY

```
int id = rs.getInt(0); // BAD CODE
```

```
ResultSet rs = stmt.executeQuery(  
    "select * from animal where name= 'Not in table'");  
rs.next();  
rs.getInt(1); // throws SQLException
```

```
ResultSet rs = stmt.executeQuery("select count(*) from animal");  
rs.getInt(1); // throws SQLException
```

```
ResultSet rs = stmt.executeQuery("select count(*) from animal");  
rs.next();  
rs.getInt(0); // throws SQLException
```

```
ResultSet rs = stmt.executeQuery("select id from animal");  
rs.next();  
rs.getInt("badColumn"); // throws SQLException
```



# RESULTSET – TYPY DANYCH

Method Name	Return Type	Example Database Type
getBoolean	boolean	BOOLEAN
getDate	java.sql.Date	DATE
getDouble	double	DOUBLE
getInt	int	INTEGER
getLong	long	BIGINT
getObject	Object	Any type
getString	String	CHAR, VARCHAR
getTime	java.sql.Time	TIME
getTimeStamp	java.sql.TimeStamp	TIMESTAMP

# RESULTSET – TYPY DANYCH

JDBC Type	Java 8 Type	Contains
<code>java.sql.Date</code>	<code>java.time.LocalDate</code>	Date only
<code>java.sql.Time</code>	<code>java.time.LocalTime</code>	Time only
<code>java.sql.Timestamp</code>	<code>java.time.LocalDateTime</code>	Both date and time

# RESULTSET – TYPY DANYCH

```
ResultSet rs = stmt.executeQuery("select date_born from animal where name = 'Elsa'");
if (rs.next()) {
    java.sql.Date sqlDate = rs.getDate(1);
    LocalDate localDate = sqlDate.toLocalDate();
    System.out.println(localDate); // 2001-05-06
}
```

```
ResultSet rs = stmt.executeQuery("select date_born from animal where name = 'Elsa'");
if (rs.next()) {
    java.sql.Time sqlTime = rs.getTime(1);
    LocalTime localTime = sqlTime.toLocalTime();
    System.out.println(localTime); // 02:15
}
```

```
ResultSet rs = stmt.executeQuery("select date_born from animal where name = 'Elsa'");
if (rs.next()) {
    java.sql.Timestamp sqlTimeStamp = rs.getTimestamp(1);
    LocalDateTime localDateTime = sqlTimeStamp.toLocalDateTime();
    System.out.println(localDateTime); // 2001-05-06T02:15
}
```

# RESULTSET

Odpowiednie tryby specyfikowane w momencie kreacji obiektu **ResultSet** dają nam szersze możliwości w kontekście poruszania się po wynikowym zbiorze danych.

```
10: Statement stmt = conn.createStatement(
11:     ResultSet.TYPE_SCROLL_INSENSITIVE,
12:     ResultSet.CONCUR_READ_ONLY);
13: ResultSet rs = stmt.executeQuery("select id from species order by id");
14: rs.afterLast();
15: System.out.println(rs.previous());    // true
16: System.out.println(rs.getInt(1));    // 2
17: System.out.println(rs.previous());    // true
18: System.out.println(rs.getInt(1));    // 1
19: System.out.println(rs.last());       // true
20: System.out.println(rs.getInt(1));    // 2
21: System.out.println(rs.first());      // true
22: System.out.println(rs.getInt(1));    // 1
23: rs.beforeFirst();
24: System.out.println(rs.getInt(1));    // throws SQLException
```

beforeFirst()	→	id integer	name character varying(255)	num_acres numeric
first()	→	1	African Elephant	7.5
last()	→	2	Zebra	1.2
afterLast()	→			

# RESULTSET

```
Statement stmt = conn.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_READ_ONLY);  
ResultSet rs = stmt.executeQuery("select id from species where id = -99");  
System.out.println(rs.first()); // false  
System.out.println(rs.last());  // false
```

# RESULTSET

absolute(0)	→	id integer	name character varying(255)	num_acres numeric
absolute(1)	→	1	African Elephant	7.5
absolute(2)	→	2	Zebra	1.2
absolute(3)	→			

absolute(-3)	→	id integer	name character varying(255)	num_acres numeric
absolute(-2)	→	1	African Elephant	7.5
absolute(-1)	→	2	Zebra	1.2

# RESULTSET

```
36: Statement stmt = conn.createStatement(  
37:     ResultSet.TYPE_SCROLL_INSENSITIVE,  
38:     ResultSet.CONCUR_READ_ONLY);  
39: ResultSet rs = stmt.executeQuery("select id from animal order by id");  
40: System.out.println(rs.absolute(2));      // true  
41: System.out.println(rs.getString("id"));  // 2  
42: System.out.println(rs.absolute(0));      // false  
43: System.out.println(rs.absolute(5));      // true  
44: System.out.println(rs.getString("id"));  // 5  
45: System.out.println(rs.absolute(-2));     // true  
46: System.out.println(rs.getString("id"));  // 4
```

```
51: Statement stmt = conn.createStatement(  
52:     ResultSet.TYPE_SCROLL_INSENSITIVE,  
53:     ResultSet.CONCUR_READ_ONLY);  
54: ResultSet rs = stmt.executeQuery("select id from animal order by id");  
55: System.out.println(rs.next());           // true  
56: System.out.println(rs.getString("id"));  // 1  
57: System.out.println(rs.relative(2));      // true  
58: System.out.println(rs.getString("id"));  // 3  
59: System.out.println(rs.relative(-1));     // true  
60: System.out.println(rs.getString("id"));  // 2  
61: System.out.println(rs.relative(4));      // false
```



# RESULTSET

Method	Description	Requires Scrollable ResultSet
<code>boolean absolute(int rowNum)</code>	Move cursor to the specified row number	Yes
<code>void afterLast()</code>	Move cursor to a location immediately after the last row	Yes
<code>void beforeFirst()</code>	Move cursor to a location immediately before the first row	Yes
<code>boolean first()</code>	Move cursor to the first row	Yes
<code>boolean last()</code>	Move cursor to the last row	Yes
<code>boolean next()</code>	Move cursor one row forward	No
<code>boolean previous()</code>	Move cursor one row backward	Yes
<code>boolean relative(int rowNum)</code>	Move cursor forward or backward the specified number of rows	Yes

# 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();
}
```

# ZAMYKANIE ZASOBÓW

Wszystkie obiekty implementujące kontrakt wyznaczony poprzez interfejsy **Connection**, **Statement** oraz **ResultSet** implementują również interfejs **AutoCloseable**. Tym samym, w ramach procedury zamknięcia zasobu możemy posłużyć się konstrukcją **try-with-resources**.

```
public static void main(String[] args) throws SQLException {  
    String url = " jdbc:derby:zoo";  
    try (Connection conn = DriverManager.getConnection(url);  
        Statement stmt = conn.createStatement();  
        ResultSet rs = stmt.executeQuery("select name from animal")) {  
  
        while (rs.next())  
            System.out.println(rs.getString(1));  
    }  
}
```

# ZAMYKANIE ZASOBÓW

**Dobłą praktyką jest zamykanie wszystkich trzech zasobów** (Connection, Statement, ResultSet), niemniej jednak **nie jest to bezwzględnie konieczne**.

- Zamknięcie zasobu Connection powoduje automatyczne zamknięcie zasobów Statement oraz ResultSet.
- Zamknięcie zasobu Statement powoduje automatyczne zamknięcie zasobu ResultSet.

Uwaga:

JDBC **automatycznie zamyka zasób ResultSet**

w momencie wykonywania kolejnego zapytania **na tym samym obiekcie Statement**.

```
String url = jdbc:derby:zoo";
try (Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select count(*) from animal")) {

    if (rs.next()) System.out.println(rs.getInt(1));

    ResultSet rs2 = stmt.executeQuery("select count(*) from animal");
    int num = stmt.executeUpdate(
        "update animal set name = 'clear' where name = 'other'");
}
```

# OBSŁUGA WYJĄTKÓW

Metody dostępne na klasie **SQLException**:

- **getMessage()** – komunikat odnoszący się do opisu błędy
- **getSqlState()** – kod błędu SQL (określony poprzez JDBC API)
- **getErrorCode()** – kod błędu związany konkretną bazą danych

```
String url = "jdbc:derby:zoo";
try (Connection conn = DriverManager.getConnection(url "));
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select not_a_column from animal")) {

    while (rs.next())
        System.out.println(rs.getString(1));

} catch (SQLException e) {
    System.out.println(e.getMessage());
    System.out.println(e.getSqlState());
    System.out.println(e.getErrorCode());
}
```

```
ERROR: column "not_a_column" does not exist
Position: 8
42703
0
```

# LOKALIZCJA

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

# LOKALIZACJA – POJĘCIA PODSTAWOWE

- **Internacjonalizacja (i18n)**– proces projektowania aplikacji umożliwiający jego adaptację w oparciu o preferencje użytkownika (związane np. obsługą języka, formatem wyświetlania daty, itd...)
- **Lokalizacja (l18n)**– określa możliwość wsparcia dla różnych lokalizacji geograficznych, politycznych, kulturowych. Można ją utożsamiać (w pewnym uproszczeniu) jako parę: (kraj, język). Umożliwia translację komunikatów dla różnych języków. Umożliwia prezentację liczb, kwot, daty i czasu w określonych formatach.





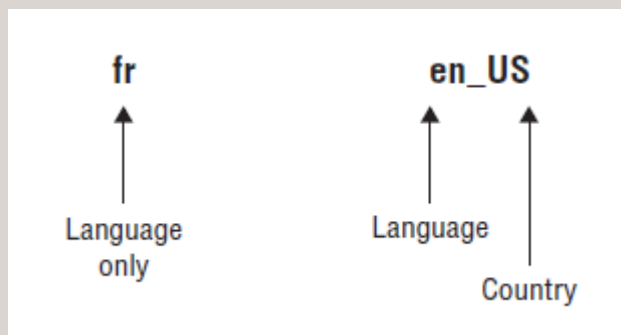
# LOCALE

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();
```

# LOCALE – MODYFIKACJA DOMYŚLNYCH WARTOŚCI

Istnieje możliwość zmiany domyślnej lokalizacji.

Uwaga. Zmiana ta dotyczy tylko i wyłącznie bieżącej aplikacji (bieżącego procesu).

```
System.out.println(Locale.getDefault()); // en_US  
Locale locale = new Locale("fr");  
Locale.setDefault(locale);              // change the default  
System.out.println(Locale.getDefault()); // fr
```



# 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

## – PLIK \*.PROPERTIES – REGUŁY

- Jeżeli linia w pliku zaczyna się od znaku **#** lub **!** oznacza to, iż jest komentarzem
- Spacje pomiędzy separatorem (**=**, **:**, **{space}**) są ignorowane
- Spacje na początku oraz końcu linii są ignorowane
- W celu złamania linii należy posłużyć się znakiem backslash (**\**)
- Istnieje możliwość wykorzystania tzw. znaków ucieczki (ang. escape characters), np.: **\t**, **\n**

```
animal=dolphin
animal:dolphin
animal dolphin
```

```
# one comment
! another comment
key =   value\tafter tab
long = abcdefghijklm\
      nopqrstuvwxyz
```

# RESOURCEBUNDLE

## – PLIK \*.PROPERTIES

W oparciu o klasę **ResourceBundle** mamy możliwość pobrania wszystkich wpisów (zestawów danych – mapowania: klucz – wartość).

```
Locale us = new Locale("en", "US");
ResourceBundle rb = ResourceBundle.getBundle("Zoo", us);

Set<String> keys = rb.keySet();
keys.stream().map(k -> k + " " + rb.getString(k))
    .forEach(System.out::println);
```

Zamiast mapy mamy również możliwość skorzystania z klasy **Properties**.

```
Properties props = new Properties();
rb.keySet().stream()
    .forEach(k -> props.put(k, rb.getString(k)));
```



Key Found?	Yes	No
getProperty("key")	Value	null
getProperty("key", "default")	Value	"default"

# 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"));
    }
}
```



# RESOURCEBUNDLE

## – KOLEJNOŚĆ PRZESZUKIWAŃ



Step	Looks for File	Reason
1	Zoo_fr_FR.java	The requested locale
2	Zoo_fr_FR.properties	The requested locale
3	Zoo_fr.java	The language we requested with no country
4	Zoo_fr.properties	The language we requested with no country
5	Zoo_en_US.java	The default locale
6	Zoo_en_US.properties	The default locale
7	Zoo_en.java	The default language with no country
8	Zoo_en.properties	The default language with no country
9	Zoo.java	No locale at all—the default bundle
10	Zoo.properties	No locale at all—the default bundle
11	If still not found, throw MissingResourceException.	



# RESOURCEBUNDLE – KOLEJNOŚĆ PRZESZUKIWAŃ

Ile plików zostanie przeszukanych w ramach poniższych instrukcji kodu?

```
Locale.setDefault(new Locale("hi"));
ResourceBundle rb = ResourceBundle.getBundle("Zoo", new Locale("en"));
```

```
Zoo.properties
name=Vancouver Zoo

Zoo_en.properties
hello=Hello
open=is open

Zoo_en_CA.properties
visitor=Canada visitor

Zoo_fr.properties
hello=Bonjour
open=est ouvert

Zoo_fr_CA.properties
visitor=Canada visiteur
```

```
Locale locale = new Locale("en", "CA");
ResourceBundle rb = ResourceBundle.getBundle("Zoo", locale);
System.out.print(rb.getString("hello"));
System.out.print(". ");
System.out.print(rb.getString("name"));
System.out.print(" ");
System.out.print(rb.getString("open"));
System.out.print(" ");
System.out.print(rb.getString("visitor"));
```

# FORMATOWANIE

W Javie istnieje wiele klas, które uwzględniają lokalizację.

Są to m.in.: **NumberFormat** (formatowanie liczb),

**DateFormat** (formatowanie dat), **Calendar** (data i czas).

Metody dostępne w klasie **NumberFormat**:

A general purpose formatter	<code>NumberFormat.getInstance()</code> <code>NumberFormat.getInstance(locale)</code>
Same as <code>getInstance</code>	<code>NumberFormat.getNumberInstance()</code> <code>NumberFormat.getNumberInstance(locale)</code>
For formatting monetary amounts	<code>NumberFormat.getCurrencyInstance()</code> <code>NumberFormat.getCurrencyInstance(locale)</code>
For formatting percentages	<code>NumberFormat.getPercentInstance()</code> <code>NumberFormat.getPercentInstance(locale)</code>
Rounds decimal values before displaying (not on the exam)	<code>NumberFormat.getIntegerInstance()</code> <code>NumberFormat.getIntegerInstance(locale)</code>

# FORMATOWANIE LICZB

```
import java.text.*;
import java.util.*;

public class FormatNumbers {
    public static void main(String[] args) {
        int attendeesPerYear = 3_200_000;
        int attendeesPerMonth = attendeesPerYear / 12;
        NumberFormat us = NumberFormat.getInstance(Locale.US);
        System.out.println(us.format(attendeesPerMonth));
        NumberFormat g = NumberFormat.getInstance(Locale.GERMANY);
        System.out.println(g.format(attendeesPerMonth));
        NumberFormat ca = NumberFormat.getInstance(Locale.CANADA_FRENCH);
        System.out.println(ca.format(attendeesPerMonth));
    } }
```



# PARSOWANIE LICZB

```
NumberFormat en = NumberFormat.getInstance(Locale.US);  
NumberFormat fr = NumberFormat.getInstance(Locale.FRANCE);  
  
String s = "40.45";  
System.out.println(en.parse(s)); // 40.45  
System.out.println(fr.parse(s)); // 40
```

```
String amt = "$92,807.99";  
NumberFormat cf = NumberFormat.getCurrencyInstance();  
double value = (Double) cf.parse(amt);  
System.out.println(value); // 92807.99
```

# FORMATOWANIE DATY I CZASU

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
DateTimeFormatter shortF = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime));      // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime));     // Jan 20, 2020 11:12:34 AM
```

# FORMATOWANIE DATY I CZASU

```
DateTimeFormatter shortDateTime =  
    DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);  
System.out.println(dateTime.format(shortDateTime));  
System.out.println(date.format(shortDateTime));  
System.out.println(time.format(shortDateTime));
```

DateTimeFormatter f = DateTimeFormatter.  (FormatStyle.SHORT );	Calling f.format (localDate)	Calling f.format (localDateTime) or (zonedDateTime)	Calling f.format (localTime)
ofLocalizedDate	Legal—shows whole object	Legal—shows just date part	Throws runtime exception
OfLocalizedDateTime	Throws runtime exception	Legal—shows whole object	Throws runtime exception
ofLocalizedTime	Throws runtime exception	Legal—shows just time part	Legal—shows whole object



DZIĘKUJĘ ZA UWAGĘ

