Języki Programowania

dr inż. Tomasz Kubik
tomasz.kubik.staff.iiar.pwr.edu.pl

Klasa `Properties`

- Klasa `Properties`
  - reprezentuje tablicę hashową (par klucz-wartość),
  - służy do zarządzania „właściwościami” (systemowymi lub użytkownika).
- Właściwość
  - para klucz-wartość, najczęściej pełni rolę jakiejś opcji
- **Właściwości systemowe** (ang. *system properties*) można ustawić:
  - poza kodem źródłowym
    - w opcjach podawanych w linii komend (np. `-Dpropertyname=value`)
    - w zmiennych środowiskowych:
      - ogólnych:
        `JAVA_TOOL_OPTIONS` (java, jar, javac) lub
        `JDK_JAVA_OPTIONS` (java)
      - specyficznych:
        `_JAVA_OPTIONS` (Oracle)
        `IBM_JAVA_OPTIONS` (IBM)
        przy czym obowiązują priorytety:
        `_JAVA_OPTIONS` (specyficzne, nadpisują inne)
        opcje podane w linii komend (z tym uruchomi się jvm)
        `JAVA_TOOL_OPTIONS` (są nadpisywana przez inne)
  - w kodzie źródłowym,
    - poprzez indywidualne wywołania

    ```
    System.setProperty(String key, String value)
    ```

    - poprzez wczytanie z pliku:

    ```
    System.getProperties().load()
    System.getProperties().loadFromXML()
    ```
- **Właściwości systemowe** można odczytać (w kodzie):

  ```
  System.getProperty(String key)
  System.getProperty(String key, String def)
  ```

Położenie plików z właściwościami

```
E:\..installation_path..\jdk-11\conf>tree /f /a
Folder PATH listing for volume ????
Volume serial number is ????-????
E:.
|   logging.properties
|   net.properties
|   sound.properties
|
+---management
|       jmxremote.access
|       jmxremote.password.template
|       management.properties
|
\---security
    |   java.policy
    |   java.security
    |
    \---policy
        |   README.txt
        |
        +---limited
        |       default_local.policy
        |       default_US_export.policy
        |       exempt_local.policy
        |
        \---unlimited
                default_local.policy
                default_US_export.policy
```

Do poczytania:
https://docs.oracle.com/javase/9/docs/api/java/util/Properties.html

Właściwości

- Właściwości **systemowe** są wbudowane w kod wirtualnej maszyny.
- Inne właściwości zwykle ładowane są z plików.

- Strumień pracy:

Klucz	Przykładowa wartość	Znaczenie
Właściwości JRE		
java.home	C:\Java\jdk11\jre	Katalog domowy jre
java.library.path	E:\Develop\lib	Ścieżka do bibliotek natywnych
java.class.path	E:\Develop\classes	Ścieżka do klas
java.ext.dirs	C:\Java\jdk11\jre\lib\ext	Ścieżka do biblitek rozszerzeń JRE
java.version	11.0.2	Wersja JDK
java.runtime.version	11.0.2	Wersja JRE
java.vendor		Dostawca jvm
java.vendor.url		URL dostawcy jvm
java.class.version		Numer wersji klas
Właściowości IO		
file.separator	\	Separator plików
path.separator	;	Separator ścieżek (np. w PATH)
line.separator	\r\n	Separator linii
Właściwości OS		
os.arch	x86	Architektura systemu
os.name	Windows 10	Nazwa systemu
os.version	10	Wersja systemu
Właściwości użytkownika		
user.dir		Bieżący katalog roboczy
user.home		Katalog domowy
user.name		Nazwa użytkownika

Zawartość `sound.properties`

```
############################################################
#               Sound Configuration File
############################################################
#
# This properties file is used to specify default service
# providers for javax.sound.midi.MidiSystem and
# javax.sound.sampled.AudioSystem.
#
# The following keys are recognized by MidiSystem methods:
#
# javax.sound.midi.Receiver
# javax.sound.midi.Sequencer
# javax.sound.midi.Synthesizer
# javax.sound.midi.Transmitter
#
# The following keys are recognized by AudioSystem methods:
#
# javax.sound.sampled.Clip
# javax.sound.sampled.Port
# javax.sound.sampled.SourceDataLine
# javax.sound.sampled.TargetDataLine
#
# The values specify the full class name of the service
# provider, or the device name.
#
# See the class descriptions for details.
#
# Example 1:
# Use MyDeviceProvider as default for SourceDataLines:
# javax.sound.sampled.SourceDataLine=com.xyz.MyDeviceProvider
#
# Example 2:
# Specify the default Synthesizer by its name "InternalSynth".
# javax.sound.midi.Synthesizer=#InternalSynth
#
# Example 3:
# Specify the default Receiver by provider and name:
# javax.sound.midi.Receiver=com.sun.media.sound.MidiProvider#SunMIDI1
#
```

Przykład odczytu/ustawienia/zapisu właściwości

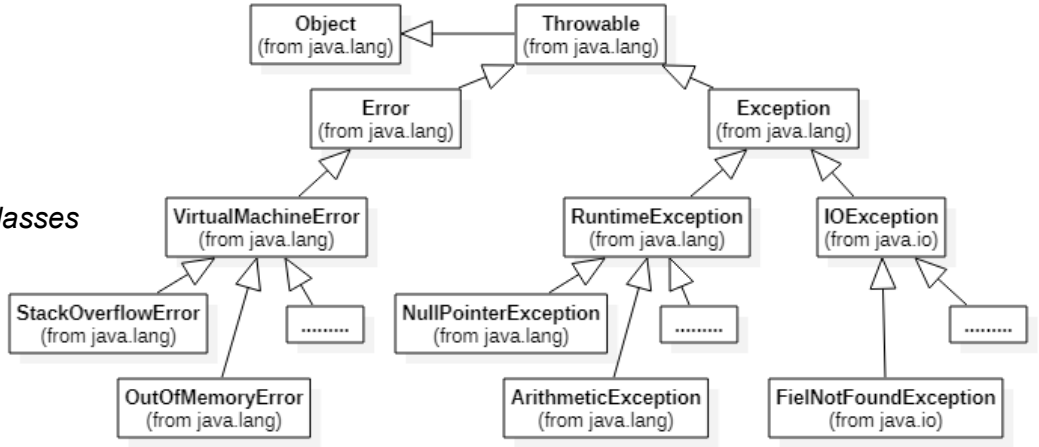
```
// Właściwości systemowe
   // Założenie strumienia na pliku "myProperties.txt"
   FileInputStream propFile = new FileInputStream("myProperties.cfg");
    // Utworzenie obiektu wypełnionego właściwościami systemowymi
    Properties p = new Properties(System.getProperties());
    // Załadowanie własnych właściwości
    p.load(propFile);
    // Ustawienie właściwości
    System.setProperties(p);
    // Wyświetlenie właściwości
    System.getProperties().list(System.out);

// Właściwości użytkownika
   // utworzenie i załadowanie właściwości domyślnych
   Properties defaultProps = new Properties();
   FileInputStream in = new FileInputStream("default.cfg");
   defaultProps.load(in);
   in.close();
   // utworzenie właściwości własnych na podstawie domyślnych
   Properties applicationProps = new Properties(defaultProps);
   // załadowanie właściwości dodatkowych
   in = new FileInputStream("appProperties.cfg");
   applicationProps.load(in);
   in.close();
   // ...
   // zapisanie właściwości
   FileOutputStream out = new FileOutputStream("appProperties.cfg");
   applicationProps.store(out, "---No Comment---");
   out.close();
```

Wyjątki

- Wyjątek jest zdarzeniem, które jeśli wystąpi podczas działania programu, to przerywa normalne jego wykonywanie (przerywa normalny strumień wykonywanych instrukcji).
- Tworzenie wyjątków i przekazywanie je do działającego programu nazywane jest **zgłaszaniem/wyrzucaniem** wyjątków (ang. *exceptions throwing*).
- Po zgłoszeniu wyjątku następuje jego **obsługa** (ang. *exceptions handling*).
- Obsługa wyjątów w Java jest strukturalna
  - przeglądane są metody na stosie wywołań, począwszy od najbardziej zagnieżdżonej, aż do znalezienia takiej, która obsługuje wyjątek.
- Mówi się, że kod obsługujący wyjątek **przechwytuje** go. Jeśli wyjątek nie zostanie przechwycony, następuje zakończenie działania programu.
- Wyjątki tworzą strukturę klas
  - Klasa `Exception` i jej klasy potomne, poza klasą `RuntimeException` i jej potomkami, nazywane są wyjątkami sprawdzalnymi/przechwytywalnymi (ang. *checked exceptions*)
    - umieszczne są w klazuli **`throws`** metod lub kontruktorów, dzięki czemu kompilator może sprawdzić, czy zgłoszenia te zostaną obsłużone (a obsłużone być muszą)
  - Klasa `RuntimeException` oraz jej klasy potomne nazywane są wyjątkami niesprawdzalnymi (ang. *unchecked exceptions*).
    - nie umieszcza się ich w klauzuli **`throws`**, kompilator nie sprawdza konieczności ich obsłużenia.

Statistics jdk 1.7:
Throwable hierarchy: 785 classes
Error hierarchy: 63 classes
Exception hierarchy: 730 classes
RuntimeException hierarchy: 242 classes



Wyjątki

- Aby zadeklarować własny wyjątek wystarczy zdefiniować klasę:

```
class MyException extends Exception { ...}
```

  Istnieją dwa standardowe konstruktory wyjątków: bez parametrów i z łańcuchem znaków. Jeśli więc łańcuch znaków ma zostać użyty jako argument konstruktora, należy w konstruktorzez własnego wyjątku wywołać konstruktor klasy nadrzędnej: `super(str);` (gdzie `str` to rzeczony łańcuch znaków).

- Dobrą praktyką jest nadawanie takich nazw wyjątkom, które mówią coś o klasie bazowej.

  `MyException` to dobra nazwa dla klasy dziedziczącej po klasie `Exception`,

  `MyError` to dobra nazwa dla klasy dziedziczącej po klasie `Error`.

- Metody zgłaszające wyjątki deklarowane są z klauzulą **`throws`**, w której może pojawić się jeden wyjątek lub cała ich lista (jeśli metoda zgłasza więcej niż jeden wyjątek):

```
void metodaZglaszajacaWyjatek() throws MyException, Exception {}
```

- Aby zgłosić wyjątek należy go najpierw utworzyć, a następnie wyrzucić poleceniem **`throw`**:

```
Exception e = new Exception("Message");
throw e;
```

- Jeśli w jakimś miejscu programu zostanie wywołana metoda zgłaszająca wyjątek, to wyjątek ten będzie się propagował poprzez zagnieżdżone wywołania aż do miejsca, w którym wystąpi blok `try-catch-finally`. Tam też, w bloku **`catch`**, wyjątek zostanie obsłużony. Nieprzechwycenie wyjątku spowoduje zatrzymanie programu (w szczególności metoda `main()` może być zadeklarowana, jako metoda zgłaszająca wyjątki – tylko wtedy wyjątków zgłoszonych przez `main()` nikt nie obsłuży). Blok **`finally`** wykonywany jest zawsze (niezależnie, czy wyjątek zostanie zgłoszony, czy też nie).

```
try {
 metodaZglaszajacaWyjatek();
} catch (Exception me ) {
  // obsługa wyjątku
} finally {
    // kod uruchamiany zawsze
}
```

```
void m1() {
    try { m2();
    } catch (Exception e) {
        // obsługa wyjątku
   }}
void m2() throws Exception {
    m3(); // tu nastąpi propagacja
}
void m3() throws Exception {
    // wyrzucenie wyjątku;
}
```

Wyjątki

- Bloki `catch()`, jeśli mają posłużyć do obsługi całej listy wyjątków, powinny tworzyć „drabinkę”, w której najpierw obsługiwany jest wyjątek najbardziej specyficzny, potem wyjątki ogólniejsze (czyli wyjątki klas bazowych powinny być obsługiwane na samym końcu).
- Gdy w drabince bloków `catch()` implementacja obsługi wyjątków powtarza się, to taki zapis można skrócić. Od jdk 1.7 wprowadzono do tego specjalną składnię:
  - w bloku `catch()` można posłużyć się listą wyjątków rozdzielanych `|`, pod warunkiem, że będą to wyjątki z różnych drzew dziedziczenia.
- Przypomnienie:
  - implementacje odziedziczonych metod zgłaszających wyjątki mogą zgłaszać wyjątki tych samych typów bądź ich specjalizacji, mogą też nie zgłaszać tych wyjątków.

```
try {
 metoda();
} catch (...) {
    // obsługa wyjątku specyficznego
} catch (...) {
    // obsługa wyjątku ogólnego
}
```

```
class E extends Exception{}
class E1 extends E{}
class E2 extends E1{}
class F extends Exception{}

interface I {
  void m(int ... argi) throws E1, F;
}
public class Wyjatki implements I{
    static Wyjatki w;

  @Override
  public void m(int ... argi) throws E1, F{
  }

  public static void main(String ... args)
              throws Throwable  {
    w = new Wyjatki();
      try {
        w.m(1);
      } //catch (Throwable t)// Dobrze
                  // (przechwycono wszystkie wyjątki)
        catch(E1 | F e)       // Dobrze
                  // (przechwycono wyjątki E1 i F)
      // catch(E1 | E | F e)  // Źle
                  // (E1 dziedziczy z E)
      {
        throw e;
      }
  }
}
```

Do poczytania:
http://tutorials.jenkov.com/java-exception-handling/exception-enrichment.html

Wyjątki

- Od jdk 1.7 w języku Java można używać bloków try-with-resource.

```
try (...) {
    ...
  } catch (...) {
    ...
    }
```

- Pozwalają one skrócić kod związany z obsługą wyjątków generowanych przez obiekty dostarczające implementacji interfejsu `AutoCloseable` (zwalniając z konieczności jawnego ich zamykania).
- To tzw. *syntatic sugar* (kompilator i tak wyprodukuje w kodzie bajtowym sekwencję `try/catch/finally`, patrz Do poczytania).
- Istnieją też zalecenia, by nie stosować tzw. antywzorów (patrz Do poczytania).

```
// kod standardowy
Scanner sc;
try {
    sc = new Scanner(new File(fileName));
    return Integer.parseInt(sc.nextLine());
} catch (FileNotFoundException ex1) {
          System.out.println("File not found");
} finally {
    try {
        if (sc != null) {
            sc.close();
        }
    } catch (IOException ex2) {
          System.out.println("Couldn't close the scanner");
    }
}
```

```
// kod skrócony
try (Scanner sc = new Scanner(new File(fileName))) {
    return Integer.parseInt(sc.nextLine());
  } catch (FileNotFoundException e ) {
    System.out.println("File not found");
    return 0;
  }
```

Do poczytania:
https://www.baeldung.com/java-exceptions
https://www.samouczekprogramisty.pl/konstrukcja-try-with-resources-w-jezyku-java/
https://stackify.com/best-practices-exceptions-java/
https://belief-driven-design.com/functional-programming-with-java-exception-handling-e69997c11d3/

Wątki

- Istnieją dwa standardowe sposoby deklarowania wątków
  - utworzenie klasy dziedziczącej po `Thread`, implementacja metody `run()` (implementacja tej metody w klasie `Thread` jest pusta), uruchomienie konstruktowa tej klasy,
  - przekazanie instancji klasy implementującej interfejs `Runnable` do konstruktora klasy `Thread` (w szczególności można posłużyć się tu wyrażeniem lambda).
- Uruchomienie wątku polega na odpaleniu jego metody `start()`
  - metodę tę można odpalić tylko raz.

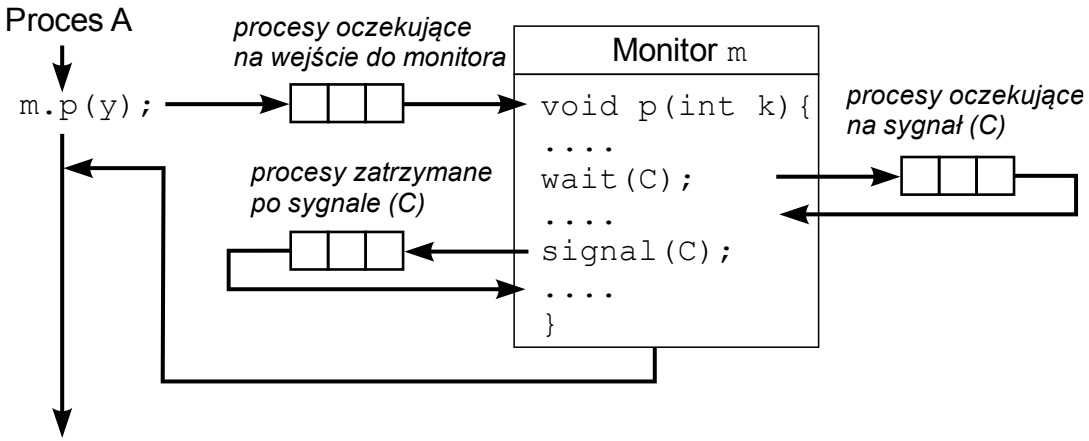
- Ponadto istnieją pakiety klas ułatwiające pisanie aplikacji wielowątkowych (pakiet `java.util.concurrent` i pakiety podległe `java.util.concurrent.atomic`, `java.util.concurrent.locks`)
  - `Executor, ExecutorService, ScheduledExecutorService, Future, CountDownLatch, CyclicBarrier, Semaphore, ThreadFactory, BlockingQueue, DelayQueue, Locks, Phases`

```
class A extends Thread {
  public A(String name) {super(name);}
  public void run() {
     System.out.println("My name is " +
getName());
  }
}
class B {
  public static void main(String[] args) {
     A a = new A("ThreadA");
     a.start();
  }
}
```

```
class C extends ...  implements Runnable {
  public void run() {
    System.out.println("My name is " +
getName());
  }
}
class B {
  public static void main(String[] args) {
     C c = new C();
     Thread t = new Thread(c, "ThreadT");
     t.start();
  }
}
```

Synchronizacja wątków

- Język Java posiada wbudowany mechanizm synchronizacji wątków, działający wg wzorca Monitora. Wykorzystuje się w nim słowo kluczowe **`synchronized`**.
  - **aby doszło do synchronizacji, obiekt synchronizacji musi być przez wątki współdzielony!!!!!**



- Istnieją dwa konteksty, w którym słowo **`synchronized`** może się pojawić:
  - przed sygnaturą metody (co równoważne jest synchronizacji po **`this`**)

```
synchronized type method ( attributes ) {
  // critical section
}
```

  - w bloku kodu (razem z nawiasami okrągłymi, w których deklaruje się obiekt synchronizacji)

```
synchronized (object) {
  //critical section
}
```

- Ważne też jest słowo **`volatile`**, dzięki któremu można wyeliminować problemy występujące w aplikacjach wielowątkowych wynikające z optymalizacji (wykorzystania pamięci podręcznej)
  - zmienna opatrzona tym słowem staje się „zapamiętana w głównej pamięci”, dzięki czemu
  - każdy odczyt takiej zmiennej odbywać się będzie z głównej pamięci komutera, a nie z pomięci podręcznej CPU oraz każdy zapis takiej zmiennej będzie odbywał się do głównej pamięci, a nie do pamięcie podręcznej CPU

Do poczytania:
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/
https://www.baeldung.com/java-volatile

Przykład

```
import java.io.IOException;

class W extends Thread{
  @Override
  public void run() {
    for(int i=0;i<5;i++) {
      System.out.println("i="+i);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();}
    }
  }

  @Override
  protected void finalize() throws Throwable {
    System.out.println("Usuwany wątek");
  }
}

public class Test {
  protected void finalize() {
    System.out.println("finalize");
  }
  public static void main(String[] args) throws IOException,
InterruptedException {
  Test t = new Test();
  t = null;
  W w = new W();
  w.start();
// Dany wątek można wystartować tylko raz.
// Ponowne wystartowanie spowoduje wyrzucenie wyjątku
java.lang.IllegalThreadStateException
  w = null;
// Wątki, do których zgubiono referencję zostaną posprzątane
// przez odśmiecacza, jednak dopiero wtedy,
// gdy zakończą działanie.
// Działający wątek tworzy tzw. garbage collection root,
// który jest "osiągalny" dopóki wątek "żyje".
// Czyli jeśli istnieje jakiś obiekt przypisany do pola wątku
// lub zmiennej, która widoczna jest w metodzie run,
// to taki obiekt nie zostanie usunięty z pamięci,
// dopóki wątek działa.

// Podczas robienia porządków odśmiecacz zaczyna sprawdzać
// osiągalność obiektów właśnie od garbage collection root.
// Jeśli dany garbage collection root przestaje być
// osiągalny, nieosiągalne stają się również
// podległe obiekty.
// Czyli po zakończeniu działania wątku, jeśli nigdzie nie ma
// zapamiętanej do niego referencji, wątek jest usuwany
// z pamięci razem z obiektami podległymi.


// Podczas debugowania można natknąć się na pewne wyjątki.
// Otóż android debugger obroni przez odśmiecaniem obiekty,
// które są obserwowane
// (obroni więc nawet wątki, które zakończyły działanie).

    System.gc();
    //System.runFinalization();
    Thread.sleep(1000);
    System.in.read();
    System.gc();
  }
}
```

Do poczytania:
https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/