

Co nowego w Javie 7

Zmiany w języku, bibliotece, narzędziach

Kamila Chyla

W skrócie

- Project "Coin" - zmiany w języku
- I/O
- Współbieżność (fork/join)
- Swing
- G1 garbage collector

Zmiany w języku

- literały liczbowe
- String w instrukcji switch
- "diamond operator"
- instrukcja try-with-resources
- "muticatch"
- bardziej precyzyjny rethrow

Literały

- binarne
- separator _
- Zastosowania:
 - implementacja protokołów
 - przesunięcia bitowe

```
byte eleven = 0b00001011;  
byte two = 0B00000010;
```

```
int mySalary = 12_860_000;  
long myDebt = 1_000_000_000L;  
float sizeofBacteria = 0.000_000_023F;
```

Diamond Operator

pre-Java7

```
<String, List<Strategy>> guide = new TreeMap<String, List<Strategy>> ();
```

```
<String, List<Strategy>> guide = new TreeMap<> ();
```

Java7

- Type inference (wnioskowanie typów)
- typ musi wynikać z kontekstu

Diamond Operator (2)

```
List<String> list = new ArrayList<>();  
list.add("A");
```

```
// fails: addAll expects  
// Collection<? extends String>
```

```
list.addAll(new ArrayList<>());
```

//compiles

```
List<? extends String> list2 =  
new ArrayList<>();
```

```
list.addAll(list2);
```

Try-with-resources

- deklaruje zasób(y)
- zapewnia zamknięcie zasobu na końcu bloku
- `java.lang.AutoCloseable`
- channels, streams, locks, -writers, -readers, sockets

```
static String readFirstLineFromFile(String
path)throws IOException {

    try (BufferedReader br = new
BufferedReader(new FileReader(path))) {

        return br.readLine();

    }
}
```

String w instrukcji switch

- Porównywanie: `String.equals`
- bardziej efektywny bytecode

String w instrukcji switch

```
void doAction(Strategy s) {  
    String strategyMarker = s.marker();  
    switch(strategyMarker) {  
        case "SIMPLE": check(s); break;  
        case "MEDIUM": accept(s); break;  
        case "COMPLEX": verify(s); break;  
    }  
}
```

"Multicatch"

- Blok "catch" obsługuje wiele wyjątków

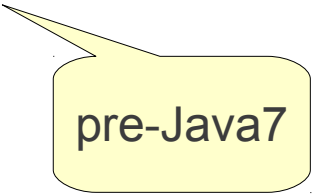
"Multicatch"

- Blok "catch" obsługuje wiele wyjątków
- Zmienna wyjątku niejawnie "final"

"Multicatch"

- Blok "catch" obsługuje wiele wyjątków
- Zmienna wyjątku niejawnie "final"

```
catch (IOException ex) {  
    logger.log(ex);  
    throw ex;  
} catch (SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```



pre-Java7

"Multicatch"

- Blok "catch" obsługuje wiele wyjątków
- Zmienna wyjątku niejawnie "final"

```
catch (IOException ex) {  
    logger.log(ex);  
    throw ex;  
}  
catch (SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

pre-Java7

```
catch (IOException|SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

Java7

Precyzyjny "re-throw"

- Lepsze sprawdzenie rzeczywistego typu wyjątku

Precyzyjny "re-throw"

- Lepsze sprawdzenie rzeczywistego typu wyjątku

```
static class FirstException extends Exception { }
```

```
static class SecondException extends Exception { }
```

```
public void rethrowException(String exceptionName) throws Exception {  
    try {  
        if (exceptionName.equals("First")) {  
            throw new FirstException();  
        } else {  
            throw new SecondException();  
        }  
    } catch (Exception e) {  
        throw e;  
    }  
}
```

Precyzyjny "re-throw"

- Lepsze sprawdzenie rzeczywistego typu wyjątku

```
static class FirstException extends Exception { }  
static class SecondException extends Exception { }  
  
public void rethrowException(String exceptionName)  
throws FirstException, SecondException {  
    try {  
        // ...  
    }  
    catch (Exception e) {  
        throw e;  
    }  
}
```


Precyzyjny "re-throw"

- Warunki na ponownie rzucony wyjątek

```
static class FirstException extends Exception { }  
static class SecondException extends Exception { }
```

```
public void rethrowException(String exceptionName)  
throws FirstException, SecondException {  
    try {  
        // ...  
    }  
    catch (Exception e) {  
        throw e;  
    }  
}
```

e w catch jest final – nie przypisujemy nic do e

nie ma wcześniejszego catch-a, który złapie ten wyjątek

jest nadtypem lub podtypem zadeklarowanego w catch typu

Nowości w bibliotece: I/O

- `java.nio.file.Path`
- `SimpleFileVisitor`
- `PathMatcher`
- `IOException`: `FileSystemException` i podklasy
- `java.nio.file.attribute`
 - `{BasicFile, AclFile, DosFile, PosixFile}AttributeView`

```
UserPrincipalLookupService lookupService = FileSystems.getDefault().getUserPrincipalLookupService();  
Files.setOwner(path, lookupService.lookupPrincipalByName("joe"));
```

Path

- rozróżnienie OS-ów
- symlinki, hardlinki
- używana w operacjach klasy `Files.newXXX`
 - `channel`, `reader`, `writer`, `directoryStream`
- globbing (składnia: `PathMatcher`)

```
Path dir = ...
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "*.java")) {
    for (Path entry: stream) {
        ...
    }
}
```

Glob

- PathMatcher FileSystem.
getPathMatcher(String syntaxAndPattern)
 - boolean matcher.matches(path)
- "syntax:pattern", syntax = glob lub regex
- składnia glob
 - *, **, ?, h[aou]me, *.{cpp, java, html}, ale . Zwykła
 - n[!o-z]tes.txt → nates.txt, netes.txt ale nie nutes.txt i notes.txt

FileVisitor

- `Files.walkFileTree(Path start,
Set<FileVisitOption> options,
int maxDepth,
FileVisitor<? super Path> visitor)`
- `Files.walkFileTree(Path start,
FileVisitor<? super Path> visitor)`
- Obydwie mogą rzucić `IOException`

FileVisitor(2)

- Interfejs:

preVisitDirectory

postVisitDirectory

visitFile

visitFileFailed

java.util.concurrent

- Fork-join: algorytmy
 - dziel i rządź / MapReduce

if (moja część jest wystarczająco mała)

Wykonaj zadanie bezpośrednio

else

Podziel część na dwie

Wykonaj zadanie dla każdej z części i czekaj na rezultaty

java.util.concurrent

- Fork-join: algorytmy
 - dziel i rządź / MapReduce
 - "map" - dzielenie przestrzeni danych
 - "reduce" - wyliczanie rezultatu z wyników częściowych

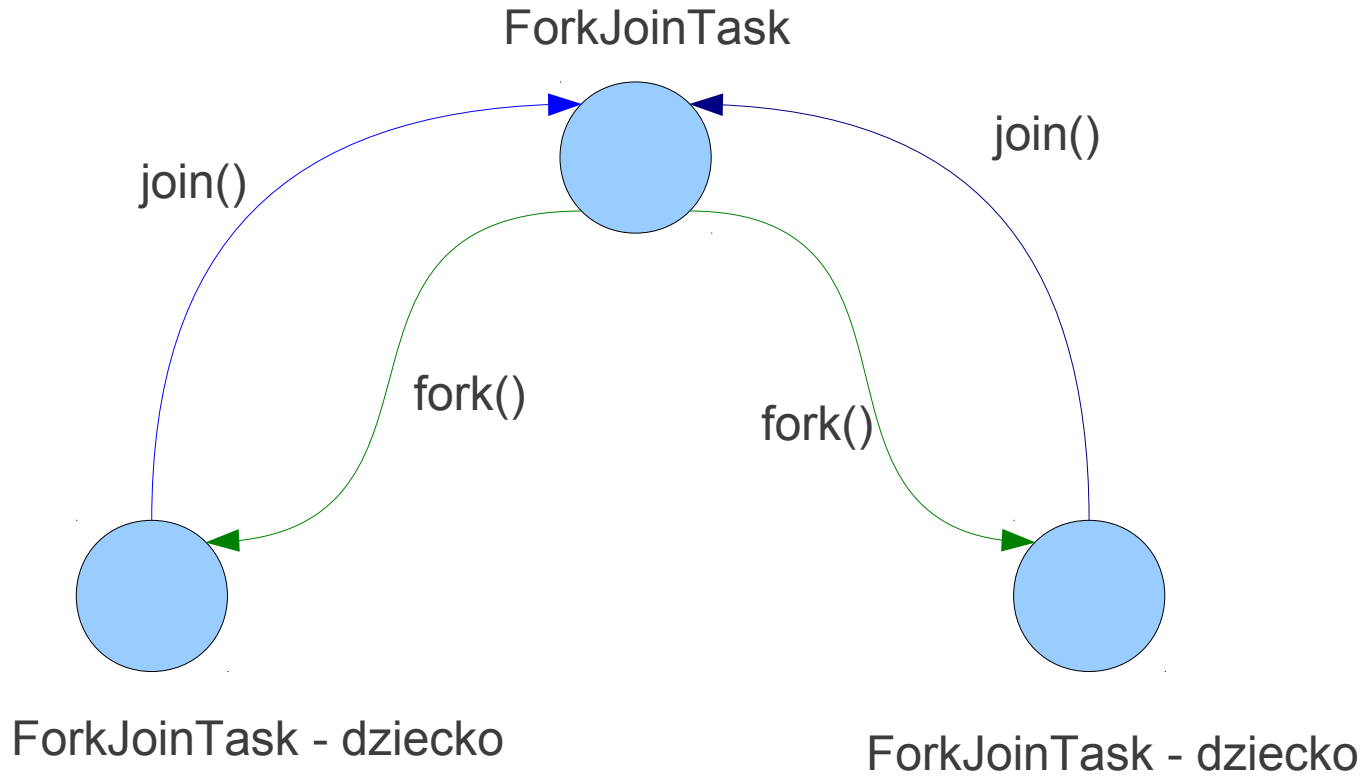
java.util.concurrent

- Fork-join: algorytmy
 - dziel i rządź / MapReduce
 - "map" - dzielenie przestrzeni danych
 - "reduce" - wyliczanie rezultatu z wyników częściowych
- Callable->Executor Service
 - charakter danych
 - implementacja wątków w os

java.util.concurrent

- ForkJoinPool: ExecutorService z "podkradaniem zadań"
- Tworzony z zadaniem poziomem równoległości
- ForkJoinTask
 - fork() - zaplanowany do wykonania asyn., "odpalony" z istniejącego zadania
 - join() - pozwala zadaniu czekać na zakończenie innego
 - RecursiveTask, RecursiveAction

fork-join



Swing

- Dekorowanie komponentów (Jlayer, LayerUI)
- Przezroczyste okna, różne kształty okien

`isWindowTranslucencySupported(GraphicsDevice.WindowTranslucency)`

TRANSLUCENT – jednolita wartość alpha ("półprzezroczyste" okna)

PERPIXEL_TRANSLUCENT – każdy piksel ma inną wartość alpha

PERPIXEL_TRANSPARENT – kształty okien

window.setOpacity(float) – TRANSLUCENT

paintComponent: `g2d.setPaint(p)`; gdzie `p` – GradientPaint - PERPIXEL_TRANSLUCENT

window.setShape(Shape) – PERPIXEL_TRANSPARENT

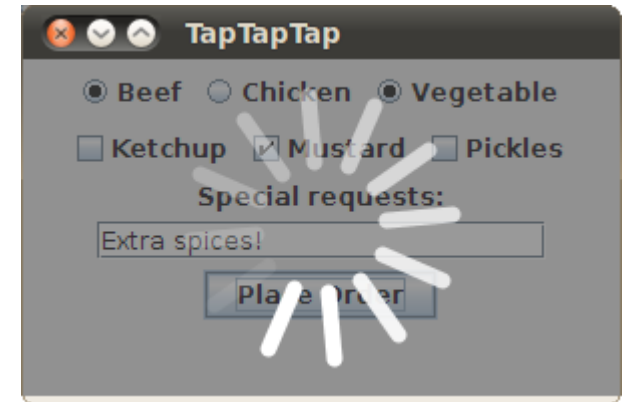
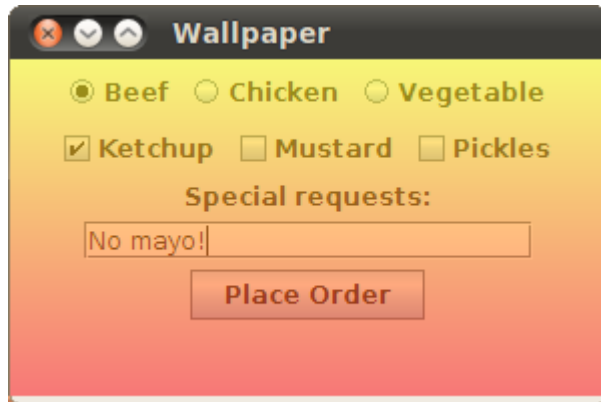
- JColorChooser: dodane CMYK, HSL

Swing (2)

- Dodana klasa `javax.swing.JLayer`
- Potrzebna `javax.swing.plaf.LayerUI`
 - Utwórz dekorowany komponent (target, np. `JButton`)
 - Stwórz podklasę `LayerUI` (tu rysujemy)
 - Stwórz obiekt `JLayer` wrapujący target i obiekt rysujący
 - Użyj komponentu `JLayer` w interfejsie graficznym

Swing (3)

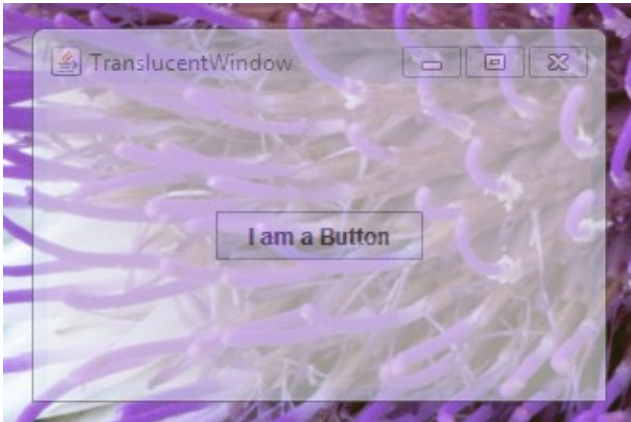
- Przykład: (<http://docs.oracle.com/javase/tutorial/uiswing/misc/jlayer.htm>)



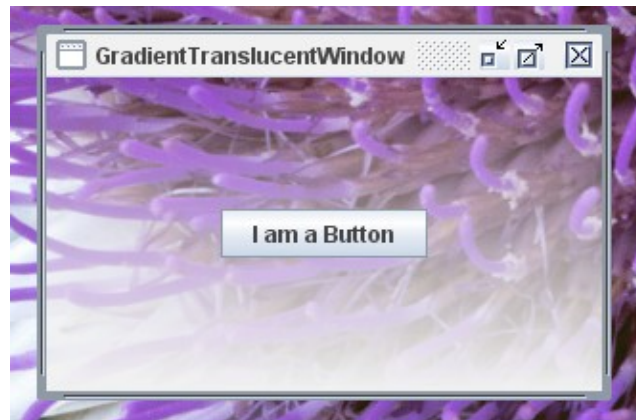
Swing (4)

- Przezroczystość: przykłady

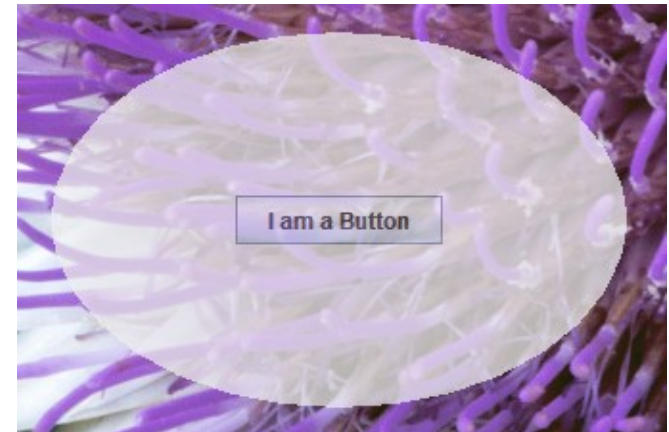
http://docs.oracle.com/javase/tutorial/uiswing/misc/trans_shaped_windows.html



uniform translucency



per-pixel translucency



per-pixel translucency - shapes

G1

- W pełni zrównoleglony, współbieżny z aplikacją
 - minimalizacja przestojów
 - zmniejszony narzut pamięciowy
 - większa przepustowość
- podział pamięci na małe fragmenty (~1MB)
- niezależne, równoległe sprzątanie
- dużo śmieci = większy priorytet (g1)

pre-G1

- Obszary "young" (eden + dwa survivor)
 - Większość żyje 10-100 mikrosekund
 - Niewiele przeżywa
 - Mało => Szybko można skopiować do survivor
 - Survivor => szybko do tenured
 - częste zbiórki "minor"
- Obszar "tenured" - full GC ("major")

G1

- Faza "mark":
 - "young" przerzucane do pustych obszarów
 - "tenured": wyznaczany "stopień żywotności" obszaru
 - Jeśli 0% - oznaczany jako wolny
- Faza "zbiórki":
 - Obszary o "niskiej żywotności" odśmiecane
 - Nie ma więc osobnej "zbiórki" dla "starych" obiektów (podziału na zbiórki typu "minor" i "major")

Koniec

- Pytania