

JAVA

Krátce o Reflection API

Přehled

- reflection, introspection
- umožňuje
 - zjišťování informací o třídách, attributech, metodách
 - vytváření objektů
 - volání metod
 - ...
- balík `java.lang.reflect`
- třída `java.lang.Class<T>`

java.lang.Class

- instance třídy **Class** reprezentuje třídu (interface, enum,...) v běžícím programu
- primitivní typy také reprezentovány jako instance třídy **Class**
- nemá žádný konstruktor
- instance vytvářeny automaticky při natažení kódu třídy do JVM
 - třídy jsou natahovány do JVM až při jejich prvním použití

java.lang.Class

- získání instance třídy **Class**
 - `getClass()`
 - metoda na třídě `Object`
 - vrátí třídu objektu, na kterém je zavolána
 - literál `class`
 - `JmenoTridy.class`
 - třída pro daný typ
 - `Class.forName(String className)`
 - statická metoda
 - vrátí třídu daného jména
 - pro primitivní typy
 - statický atribut `TYPE` na wrapper třídách
 - `Integer.TYPE`
 - literál `class`
 - `int.class`

java.lang.Class

- třídy do JVM natahuje *classloader*
 - `java.lang.ClassLoader`
 - standardní classloader hledá třídy v CLASSPATH
 - lze si napsat vlastní classloader
 - `Class.forName(String className, boolean initialize, ClassLoader cl)`
 - natáhne třídu daným classloaderem a vrátí objekt třídy `Class`
 - `getClassLoader()`
 - metoda na `Class`
 - classloader, kterým byla třída natažena

java.lang.Class: metody

- `String getName()`
 - vrátí jméno třídy
 - pro primitivní typy vrátí jeho jméno
 - pro pole vrátí řetězec začínající znaky [(tolik, kolik má pole dimenzí) a pak označení typu elementu
Z..boolean, B..byte, C..char, D..double, F..float, I..int, J..long, S..short, Lclassname..třída nebo interface

```
String.class.getName() // vrátí "java.lang.String"  
byte.class.getName() // vrátí "byte"  
(new Object[3]).getClass().getName()  
// vrátí "[Ljava.lang.Object;"  
(new int[3][4][5][6][7][8][9]).getClass().getName()  
// vrátí "[[[[[[[[I"
```

java.lang.Class: metody

- `public URL getResource(String name)`
- `public InputStream getResourceAsStream(String name)`
 - načte nějaký „zdroj“
 - obrázky,, cokoliv
 - data načítá classloader => načítání se řídí stejnými pravidly jako načítání tříd
 - jméno „zdroje“ ~ hierarchické jméno jako u tříd
 - oddělovací tečky jsou nahrazeny lomítky ' / '

java.lang.Class: metody

- **is... metody**
 - `boolean isEnum()`
 - `boolean isInterface()`
 - ...
- **get... metody**
 - `Field[] getFields()`
 - `Method[] getMethods()`
 - `Constructor[] getConstructors()`
 - ...
- ...

Použití Reflection API

- informace o kódu
 - dynamické načítání
 - pluginy
 - proxy třídy
 - ...
-
- podrobně v LS

JAVA

jar

Přehled

- vytváření archivů sdružujících .class soubory
- **JAR** ~ **J**ava **A**rchive
- soubor
 - přípona .jar
 - formát – ZIP
 - soubor META-INF/MANIFEST.MF
 - popis obsahu
- použití – distribuce softwaru
 - do CLASSPATH lze psát .jar soubory
 - lze přímo spouštět .jar soubory
- nemusí obsahovat jen .class soubory
 - obrázky
 - audio
 - cokoliv

Použití

- vytvoření archivu

```
jar cf soubor.jar *.class
```

- vytvoří soubor.jar se všemi .class soubory
- přidá do něj MANIFEST.MF soubor

```
jar cmf manifest soubor.jar *.class
```

- vytvoří soubor.jar s daným MANIFEST souborem

```
jar cf0 soubor.jar *.class
```

- nepoužije se komprese
- pro další parametry viz dokumentaci

- práce s jar archivy v programu

- java.util.jar, java.util.zip

MANIFEST.MF soubor

- seznam dvojic
 - jméno : hodnota
 - inspirováno standardem RFC822
- dvojice lze seskupovat do skupin
 - skupinu odděleny prázdným řádkem
 - hlavní skupina (první)
 - skupiny pro jednotlivé položky archivu
- délka řádků max. 65535
- konce řádků
 - CR LF, LF, CR

MANIFEST.MF soubor

- hlavní sekce
 - Manifest-Version
 - Created-By
 - Signature-Version
 - Class-Path
 - Main-Class
 - aplikace lze spouštět
java -jar archiv.jar
- vedlejší sekce
 - první položka
Name: cesta_k_položce_v_archivu

Jar a Ant

- task **jar** v Antu

- parametry

- destfile, basedir, includes, excludes, manifest

- vnořené elementy

- manifest

- příklady

```
<jar destfile="${dist}/lib/app.jar"  
      basedir="${build}/classes"  
      excludes="**/Test.class"  
    />
```

```
<jar destfile="test.jar" basedir=".">  
  <include name="build"/>  
  <manifest>  
    <attribute name="Built-By" value="${user.name}"/>  
    <section name="common/class1.class">  
      <attribute name="Sealed" value="false"/>  
    </section>  
  </manifest>  
</jar>
```

java.util.jar

- podobné jako java.util.zip
- `JarInputStream`, `JarOutputStream`
 - potomci `ZipInputStream` a `ZipOutputStream`
 - `JarInputStream` má navíc metody `getManifest()`
- `JarEntry`
 - potomek `ZipEntry`
 - získávání atributů
- `Manifest`
 - reprezentace `MANIFEST.MF` souboru

Java

Moduly

Modularizace

- modul
 - explicitně definované co poskytuje i co ***požaduje***

- proč
 - koncept *classpath* je „křehký“
 - chybí zapouzření

Modulární aplikace – motivace

- proč
 - aplikace více a více komplexní
 - skládání aplikací
 - vývoj v distribuovaných týmech
 - komplexní závislosti
 - dobrá architektura programu
 - ví o svých závislostech
 - spravuje závislosti

Deklarace modulu

- module-info.java

```
module com.foo.bar {  
    requires com.foo.baz;  
    exports com.foo.bar.alpha;  
    exports com.foo.bar.beta;  
}
```

- modular artifact

- modulární JAR – JAR obsahující module-info.class
- nový formát JMOD
 - ZIP s třídami, nativním kódem, konfigurací,...

Moduly a JDK

- standardní knihovna JDK také modulární
 - java.base – vždy „required“

```
module java.base {  
    exports java.io;  
    exports java.lang;  
    exports java.lang.annotation;  
    exports java.lang.invoke;  
    exports java.lang.module;  
    exports java.lang.ref;  
    exports java.lang.reflect;  
    exports java.math;  
    exports java.net;  
    ...  
}
```

Module readability & module path

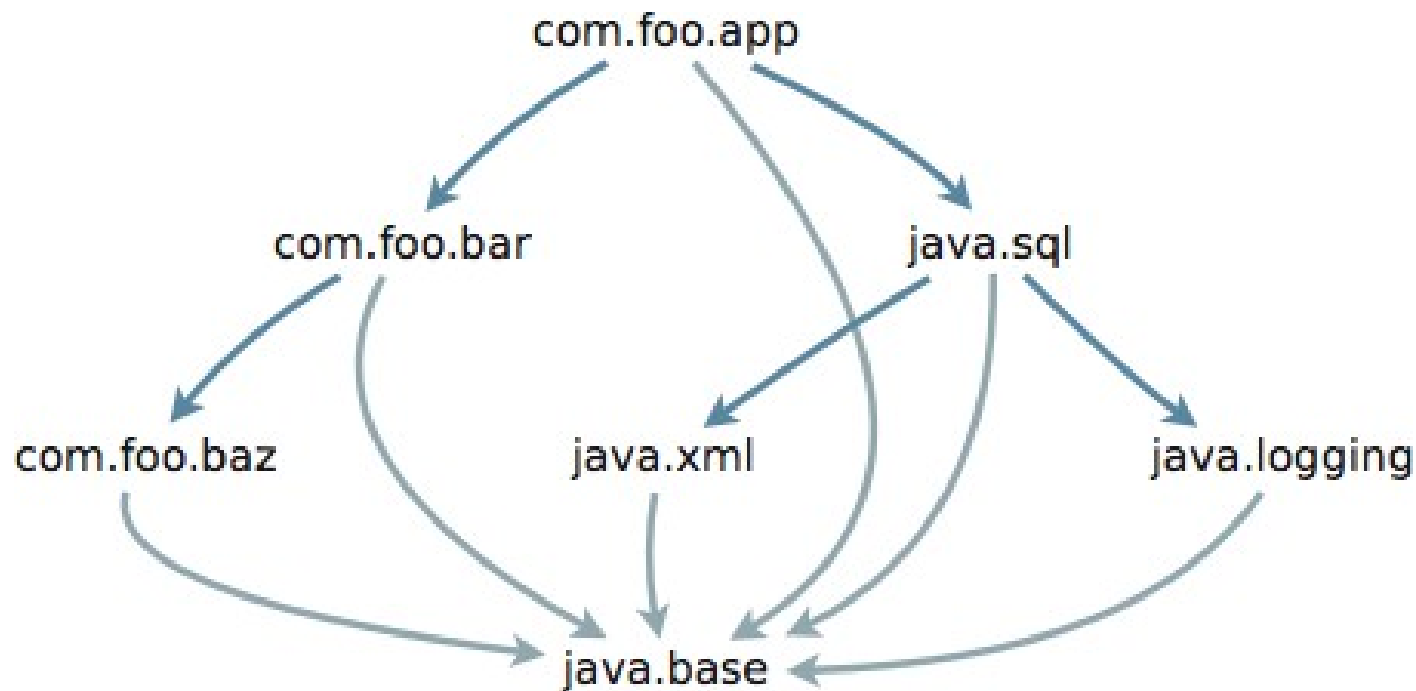
- Pokud modul přímo závisí na jiném modulu

Modul **čte** (*reads*) jiný modul (nebo, jinak, druhý modul je **čitelný** (*readable*) prvním modulem)

- **Module path** – ekvivalent ke classpath
 - ale pro moduly
 - -p, --module-path

Module graph

```
module com.foo.app {  
    requires com.foo.bar;  
    requires java.sql;  
}
```



Kompatibilita se „starou“ Javou

- Classpath stále podporováno
 - v podstatě jsou moduly „volitelné“
- Nepojmenovaný modul
 - cokoliv mimo jakýkoliv modul
 - „starý“ kód
 - čte jakýkoliv jiný modul
 - exportuje všechny svoje balíčky pro všechny jiné moduly

Moduly

- podrobně v LS

JAVA

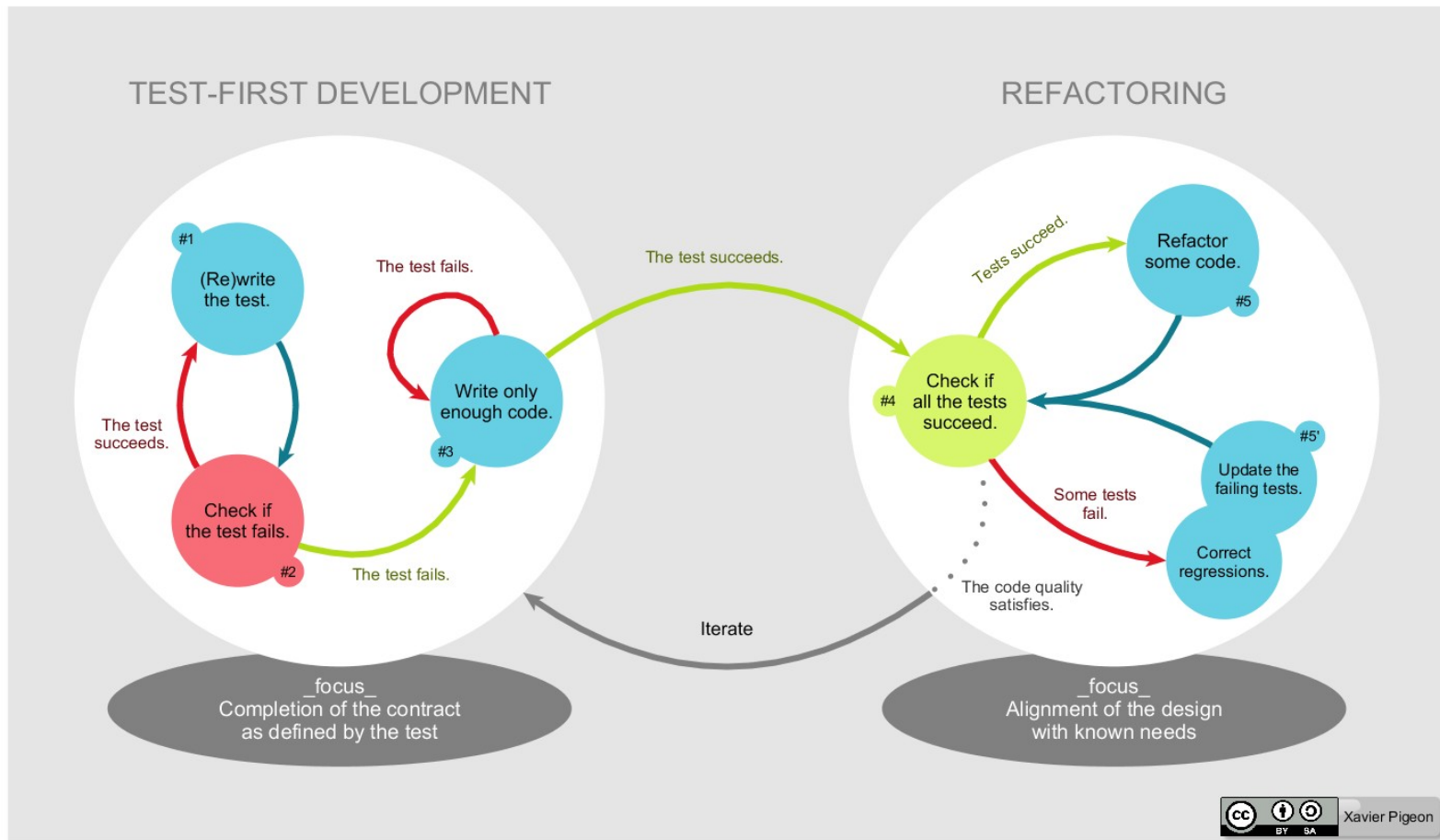
Unit testing

Úvod

- unit testing
 - testování „malý“ jednotek funkčnosti
 - jednotka – nezávislá na ostatních
 - testování zcela oddělené
 - vytvářejí se pomocné objekty pro testování
 - kontext
 - „typicky“ v OO jazycích
 - jednotka ~ metoda
 - ideálně – unit testy pro všechny jednotky v programu
 - „typicky“ v OO jazycích
 - pro všechny public metody

Test-driven development

- „začít s testy“



Zdroj: https://commons.wikimedia.org/wiki/File:TDD_Global_Lifecycle.png#/media/File:TDD_Global_Lifecycle.png

JUnit

- podpora pro unit testing v Javě
- <http://www.junit.org/>
- používání založeno na anotacích
 - starší verze založeny na dědičnosti a jmenných konvencích
- mírně odlišné použití u různých verzí
 - 5, 4, 3

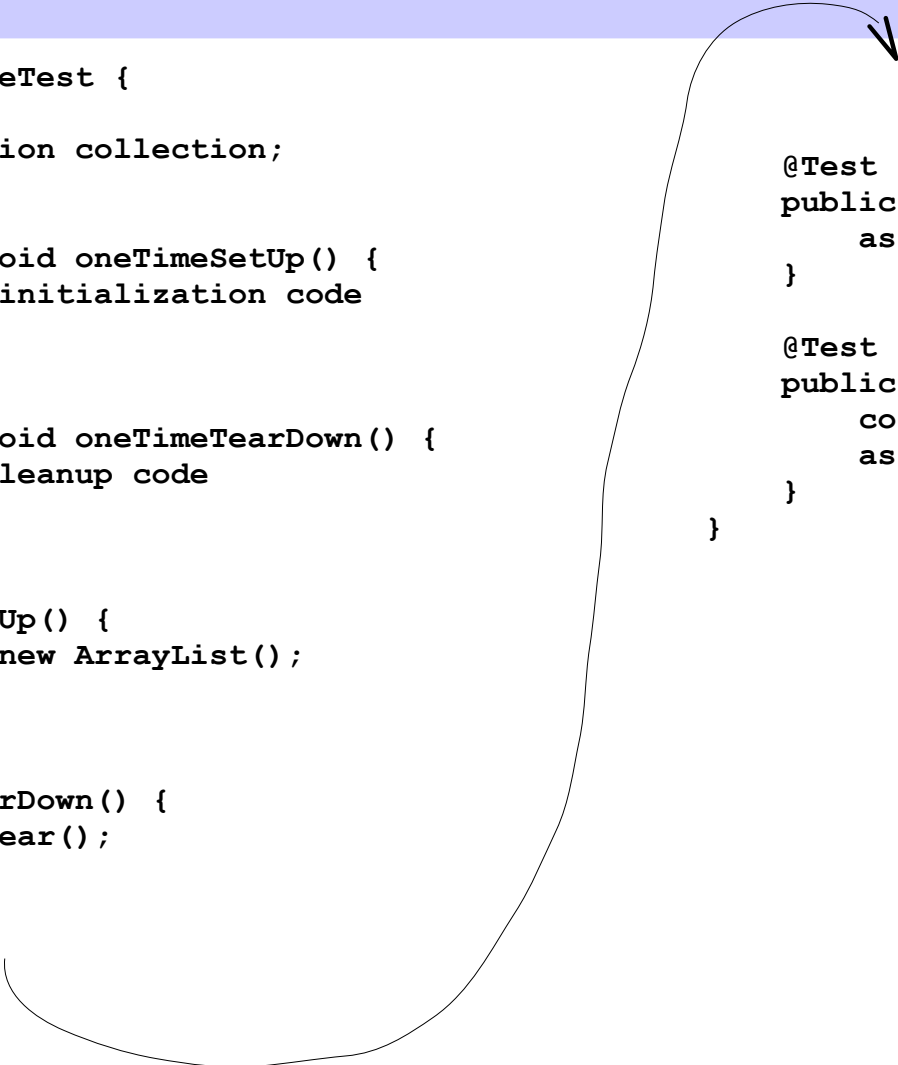
Používání

- testovací metody označeny anotací @Test
- JUnit se spustí na množinu tříd
 - najde v nich metody označené @Test
 - ty provede
- další anotace
 - @BeforeEach (@Before)
 - metoda prováděná před každým testem
 - určeno pro přípravu „prostředí“
 - @AfterEach (@After)
 - metoda provádění po každém testu
 - určeno pro „úklid“
 - @BeforeAll (@BeforeClass)
 - metoda prováděná před všemi testy na dané třídě
 - @AfterAll (@AfterClass)
 - metoda prováděná po všech testech na dané třídě

Příklad

```
public class SimpleTest {  
  
    private Collection collection;  
  
    @BeforeAll  
    public static void oneTimeSetUp() {  
        // one-time initialization code  
    }  
  
    @AfterAll  
    public static void oneTimeTearDown() {  
        // one-time cleanup code  
    }  
  
    @BeforeEach  
    public void setUp() {  
        collection = new ArrayList();  
    }  
  
    @AfterEach  
    public void tearDown() {  
        collection.clear();  
    }  
}
```

```
    @Test  
    public void testEmptyCollection() {  
        assertTrue(collection.isEmpty());  
    }  
  
    @Test  
    public void testOneItemCollection() {  
        collection.add("itemA");  
        assertEquals(1, collection.size());  
    }  
}
```



Assert

- assertTrue
- assertFalse
- assertEquals
- assert...
 - statické metody na org.junit.jupiter.api.Assertions (org.junit.Assert)
 - ověřování podmínek v testech
 - test selže pokud assert... selže
 - assert...() vyhodí AssertionError
- obecně
 - test je splněn, pokud metoda skončí normálně
 - test není splněn, pokud metoda vyhodí výjimku

Testování výjimek

- jak otestovat „správné“ vyhazování výjimek?

```
assertThrows (IndexOutOfBoundsException.class, () -> {  
    new ArrayList<Object>().get(0);  
});
```

- ve starších verzích

```
@Test(expected= IndexOutOfBoundsException.class) public void empty() {  
    new ArrayList<Object>().get(0);  
}
```

Spouštění testů

- z kódu

```
org.junit.runner.JUnitCore.runClasses (TestClass1.class, ...);
```

- z příkazové řádky

```
java -jar junit.jar -select-class TestClass1
```

- z Antu

- task junit

```
<junit printsummary="yes" fork="yes" haltonfailure="yes">  
  <formatter type="plain"/>  
  <test name="my.test.TestCase"/>  
</junit>
```

- z Mavenu

- mvn test

- z IDE

TestNG

- <http://testng.org/>
- inspirováno JUnit frameworkem
- mírně jiné vlastnosti
 - původně
 - dnes skoro stejné
- základní použití je stejné

Java

Reactive programming

Reactive programming (RP)

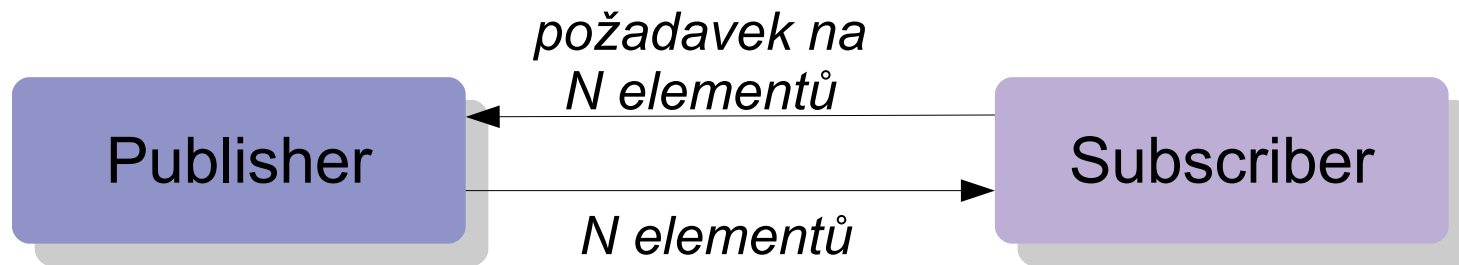
- datové toky a šíření změn v programu
 - změny v datech se automaticky propagují
- publisher-subscriber
 - architektonický vzor
 - jeden z konkrétních modelů pro RP
 - publisher produkuje data
 - subscriber asynchronně data zpracovává
 - mezi P a S mohou být procesory transformující data



- proč RP
 - jednodušší kód, efektivnější, ...
 - „rozšíření“ stream API

Publisher-Subscriber v Javě

- Flow API (Reactive streams)
- `java.util.concurrent.Flow`
 - od Java 9



- „kombinace iterátoru a Observer vzoru“

Flow API

```
@FunctionalInterface
public static interface Flow.Publisher<T> {
    public void subscribe(Flow.Subscriber<? super T> subscriber);
}

public static interface Flow.Subscriber<T> {
    public void onSubscribe(Flow.Subscription subscription);
    public void onNext(T item) ;
    public void onError(Throwable throwable) ;
    public void onComplete() ;
}

public static interface Flow.Subscription {
    public void request(long n);
    public void cancel() ;
}

public static interface Flow.Processor<T,R> extends
    Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

Flow API

- SubmissionPublisher
 - implementuje Publisher interface
 - asynchronně publikuje dodaná data
 - konstruktor bez parametrů
 - používá `ForkJoinPool.commonPool()`
 - další konstruktory – parametr pro exekutor
 - metody
 - `subscribe(Flow.Subscriber<? super T> subscriber)`
 - `submit(T item)`
 - ...

Observer pattern

- objekt (observer) „sleduje“ druhý objekt (observable)
 - pokud se druhý objekt změní, upozorní všechny „sledovatele“
 - java.util.Observer
 - java.util.Observable
 - pozor – od Java 9 jsou Deprecated (nahrazeny Flow)



- použití
 - UI
 - Observable – UI komponenty
 - Observer – reakce na UI události

Java

Ještě k vláknům

ThreadLocal

- vlastní kopie pro každé vlákno
- obvykle se používají jako statické atributy
- metody

```
T get()
protected T initialValue()
void remove()
void set(T value)
static <S> ThreadLocal<S> withInitial(Supplier<?
                                extends S> supplier)
```

Java

Pokračování...

Kam dál...

- **NPRG021 Pokročilé programování na platformě Java**
 - LS 2/2
 - obsah
 - GUI (Swing, JavaFX)
 - Moduly, Reflection API, Classloaders, Security
 - Generické typy, anotace
 - RMI
 - JavaBeans
 - Java Enterprise Edition: EJB, Servlets, Java Server Pages, Spring,...
 - Java Micro Edition: Java pro mobilní a Embedded systémy, CLDC, MIDP, MEEP
 - RTSJ, Java APIs for XML, JDBC, JMX,...
 - Kotlin a další „Java“ jazyky
 - Android
 - částečně povinné pro **NPRG059 Praktikum z pokročilého objektového programování**
 - povinný předmět pro některé Mgr. okruhy



Verze prezentace J12.cz.2019.01

Tato prezentace podléhá licenci [Creative Commons Uved'te autora-Neužívejte komerčně 4.0 Mezinárodní License](https://creativecommons.org/licenses/by-nc/4.0/).