Pokročilé programování na platformě Java

Úvod

O předmětu

- Petr Hnětynka
  - hnetynka@d3s.mff.cuni.cz
- http://d3s.mff.cuni.cz/teaching/vsjava/
- pokračování "Javy (NPRG013)"
  - předpokládá se základní znalost Javy (v rozsahu NPRG013)
- 2/2 Zk/Z

Zkouška/Zápočet

- zkouška
  - písemka
    - jako v zimním semestru
- zápočet
  - zápočtový program
    - viz další slide
  - zápočtový test
    - praktický test v labu
  - 3 domácí úkoly
    - aspoň 150 bodů
  - docházka na cvičení
    - více než 3 absence => aspoň 210 bodů z domácích úkolů

Zápočet

- napsání zápočtového programu
  - domluvit si zadání do pátku 22. května 2020
    - poslat mailem
    - přiměřeně složité téma
    - netriviálním způsobem využívat některou z technologií probíraných v předmětu
  - odevzdat nejlépe do konce června
    - nejpozději do září – úterý 22. září 2020
    - odevzdání buď emailem nebo osobně (pokud to bude nutné)

Přibližná osnova předmětu

- GUI
- hlubší pohled do jazyka Java
  - přehled a historie platformy Java
  - reflection API
  - generické typy, anotace
  - class loaders, security
- distribuované technologie: RMI,...
- komponentový model JavaBeans
- JEE: Servlety, EJB, Spring,...
- JME: CLDC, MIDP, MEEP
- RTSJ
- další technologie založené na platformě Java: Java APIs for XML, JDBC, JMX,…
- další jazyky kompilované do Java byte-code
- Android

JAVA

Java

Popularita

Feb 2020	Feb 2019	Change	Programming Language	Ratings	Change
1	1		Java	17.358%	+1.48%
2	2		C	16.766%	+4.34%
3	3		Python	9.345%	+1.77%
4	4		C++	6.264%	-1.28%
5	7	↑	C#	5.927%	+3.08%
6	5	↓	Visual Basic .NET	5.862%	-1.23%
7	6	↓	JavaScript	2.060%	-0.79%
8	8		PHP	2.018%	-0.25%
9	9		SQL	1.526%	-0.37%
10	20	⇈	Swift	1.460%	+0.54%

http://www.tiobe.com/tiobe_index

Worldwide, Feb 2020 compared to a year ago:

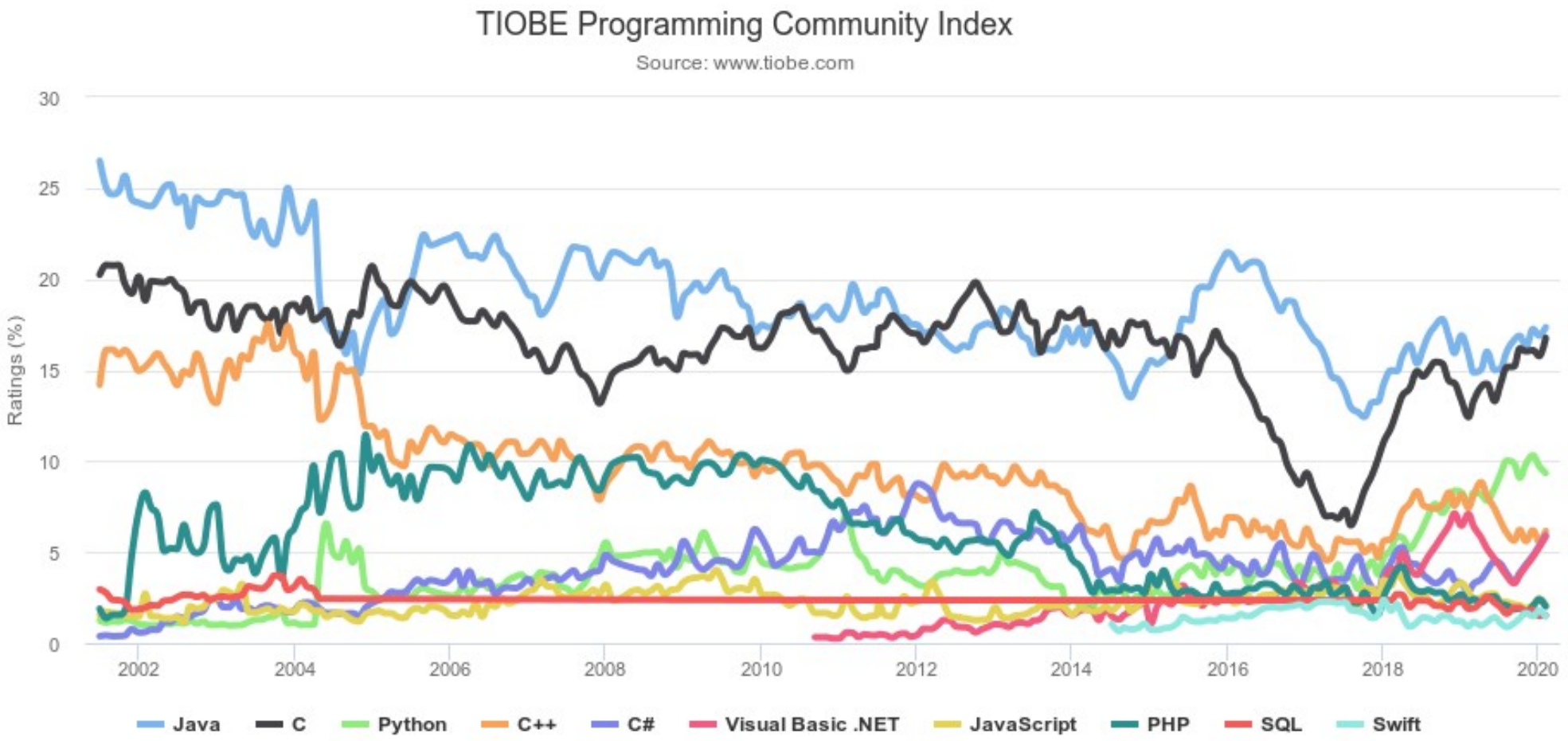
Rank	Change	Language	Share	Trend
1		Python	29.88 %	+4.1 %
2		Java	19.05 %	-1.8 %
3		Javascript	8.17 %	+0.1 %
4		C#	7.3 %	-0.1 %
5		PHP	6.15 %	-1.0 %
6		C/C++	5.2 %	-0.2 %
7		R	3.74 %	-0.2 %
8		Objective-C	2.42 %	-0.6 %
9		Swift	2.28 %	-0.2 %
10	↑	TypeScript	1.84 %	+0.3 %

http://pypl.github.io/PYPL.html

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

Popularita



zdroj: http://www.tiobe.com/tiobe_index

Typový systém

- silně typovaný jazyk
  - třídy (class)
  - primitivní typy (int, boolean, char,...)
- "vše" je v nějaké třídě
- neexistují globální proměnné, funkce,...
  - obdoba jsou **static** metody a atributy

Test

```
public class InitTest {
  static int i = 1;
  { i+=2; };
  static { i++; };
  public static void main(String argv[]) {
    System.out.println(i);
    System.out.println(new InitTest().i);
  };
};
```

Program vypíše:

a) 2 4
b) 1 3
c) 3 5
d) 4 4
e) nelze přeložit

Řešení testu

- správně je a) 2 4

- { ..... } v definici třídy
  - inicializátor
  - provádí se při vytváření nové instance
  - používá se pro inicializaci anonymních vnitřních tříd
- static { ..... }
  - statický inicializátor
  - provádí se při "natažení" třídy do VM
  - může pracovat pouze se **static** prvky třídy

Test 2

```
public class InitTest {
  static int i = 1;
  { i+=2; };
  public InitTest() {
    System.out.println(i++);
  }
  static { i++; };
  public static void main(String argv[]) {
    System.out.println(i);
    System.out.println(new InitTest().i);
  };
};
```

Výsledek:

a) 1 3 5
b) 2 3 5
c) 2 4 5

Řešení testu 2

- správně je C) 2 4 5
- inicializátor se provádí před konstruktorem
- nejdříve se inicializuje předek
  - inicializátory a konstruktor

Výjimky a inicializátory

- inicializátory mohou "vyhazovat" pouze výjimky, které jsou definovány v konstruktorech
  - musí být definován alespoň jeden konstruktor

- inicializátory anonymních tříd mohou vyhazovat jakékoliv výjimky
  - třída se vytváří pouze jednou
  - není problém výjimku odchytit / deklarovat

Statické inicializátory

- musí skončit normálně
  - jinak nelze přeložit
- prováděny v pořadí podle výskytu ve zdrojovém souboru
- nemohou obsahovat **return**
  - nelze přeložit

Viditelnost ve třídách

- lze změnit viditelnost elementů v potomcích?
  - např.

```
class A { public void foo() {} }
class B extends A { private void foo() {} }
```

- viditelnost lze „zmírnit“ ale nelze „zpřísnit“
- proč
  - pokud by bylo

```
class A { public void foo() {} }
class B extends A { private void foo() {} }
```

  - pak by šlo následující

```
A a = new B();
a.foo();
```

Typové změny

- covariantní změna – ze specifického na obecnější
- contravariantní – obráceně

- pole v Javě jsou kovariantní

```
Number[] numbers = new Number[3];
numbers[0] = new Integer(10);
numbers[1] = new Double(3.14);
numbers[2] = new Byte(0);

Integer[] myInts = {1, 2, 3, 4};
Number[] myNumbers = myInts;

Object obj = myNumbers;
```

- co se stane, pokud zkusíme následující?

```
myNumbers[0] = 3.14;
```

Covariance

- **`myNumber[0] = 3.14;`**
  - lze přeložit
  - výjimka za běhu

Reflection API

Přehled

- Reflection
  - Mění strukturu/stav objektů
- Introspection
  - Zkoumá strukturu objektů

- umožňuje
  - zjišťování informací o třídách, atributech, metodách
  - vytváření objektů
  - volání metod
  - ...
- balík `java.lang.reflect`
- třída `java.lang.Class`

java.lang.Class<T>

- instance třídy **Class** reprezentuje třídu nebo interface 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
- od Java 5 generický typ
  - T – typ třídy reprezentované instancí **Class**
    - př. pro **`String`** ~ **`Class<String>`**
    - pokud nelze určit, tak **`Class<?>`**

java.lang.Class<T>

- získání instance třídy **Class**
  - `getClass()`
    - metoda na třídě Object
    - vrátí třídu objektu, na kterém je zavolána
  - literál class (představuje *expression* typu *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<T>

- typ po získání instance

```
String s = "hello";
Class<String> clazz1 = s.getClass();

Class<String> clazz2 = String.class;

Class<Integer> clazz3 = int.class;
```

  - ale

```
Class<?> clazz4 =
            Class.forName("mypackage.MyClass");
```

java.lang.Class<T>

- 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
      - *upozornění*: typ objektu je reprezentována nejen pomocí Class, ale také classloaderu, který danou class nahrál
        - podrobně bude později

java.lang.Class<T>: metody

- `boolean isPrimitive()`
- `boolean isArray()`
- `boolean isInterface()`
- `boolean isEnum()`
- `boolean isAnnotation()`
  - test, zda třída reprezentuje primitivní typ resp. pole resp. interface resp. enum resp. anotace
- `boolean isInstance(Object o)`
  - test, zda daný objekt je instancí třídy
  - ekvivalent k operatoru instanceof
- `boolean isAssignableFrom(Class<?> cls)`
  - test, zda tato třída/interface je stejná nebo nadtřída/nadinterface `cls`
  - tj. zda objekt typu cls lze přiřadit do proměnné typu třídy, na níž je metoda volaná

java.lang.Class<T>: metody

- `String getName()`
  - vrátí jméno třídy (interfacu,...)
  - 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<T>: metody

- `Package getPackage()`
  - vrátí balík, ve kterém je třída definovaná
  - java.lang.Package
    - informace o balíku
- `Class<? super T> getSuperclass()`
  - vrátí předka třídy
  - pro třídu Object, primitivní typy a interface vrací null
- `Class<?>[] getInterfaces()`
  - vrátí všechny implementované interface
  - pokud třída neimplementuje žádný inteface, vrací pole délky 0
  - pro primitivní typy vrací také pole délky 0

java.lang.Class<T>: metody

- `Method[] getMethods()`
  - vrátí všechny metody třídy (public)
- `Field[] getFields()`
  - vrátí všechny atributy třídy (public)
- `Constructor<?>[] getConstructors()`
  - vrátí všechny konstruktory (public)
- `Method[] getDeclaredMethods()`
- `Fields[] getDeclaredFields()`
- `Constructor<?>[] getDeclaredConstructors()`
  - vrátí všechny metody/atributy/konstruktory deklarované ve třídě – nevrátí zděděné prvky

java.lang.Class<T>: metody

- `Field getField(String name)`
- `Field getDeclaredField(String name)`
  - vrátí atribut daného jména
- `Method getMethod(String name, Class<?>... paramTypes)`
- `Method getDeclaredMethod(String name, Class<?>... paramTypes)`
  - vrátí metodu daného jména a daných typů parametrů
- `Constructor<T> getConstructor(Class<?>... paramTypes)`
- `Constructor<T> getDeclaredConstructor(Class<?>... paramTypes)`
  - vrátí konstruktor s danými typy parametrů

java.lang.Class<T>: metody

- `Class<?> getDeclaringClass()`
  - vrátí třídu nebo interface, ve kterém je třída/interface deklarována
  - pro vnitřní třídy
- `Class<?>[] getClasses()`
  - vrátí třídy/interface deklarované ve třídě nebo nadtřídách
- `Class<?>[] getDeclaredClasses()`
  - vrátí třídy/interface deklarované ve třídě
- `Class<?> getComponentType()`
  - vrátí typ elementů pole
  - pokud objekt není pole, vrací null

java.lang.Class<T>: 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 `'/'`
- `T cast(Object o)`
  - od Java 5
  - v JDK 1.4 a níže by neměla význam
- `T[] getEnumConstants()`
  - vrací pole konstant enumu
    - pokud nereprezentuje enum, vrací null

java.lang.Class<T>: instance

- `T newInstance()`
  - vytvoří novou instanci dané třídy
  - použije se konstruktor bez parametrů
    - jako by se použilo new Trida()
  - **deprecated** od Java 9
    - náhrada `getDeclaredConstructor().newInstance()`

- vytváření nových instancí od třídy pomocí jiných konstruktorů
  - třída `java.lang.reflect.Constructor<T>`

Modifikátory

- `int getModifiers()`
  - metoda na java.lang.Class
  - vrátí modifikátory zakódované do integeru

- `java.lang.reflect.Modifiers`
  - dekódování integeru s modifikátory
  - statické metody
    - `boolean isPublic(int mod)`
    - `boolean isStatic(int mod)`
    - `boolean isSynchronized(int mod)`
    - `....`
    - `void toString(int mod)`
      - vrátí řetězec s modifikátory v integeru

java.lang.reflect.Field

- informace o atributech
- přístup k atributu
- metody
  - `String getName()`
    - jméno atributu
  - `Class<?> getType()`
    - typ atributu
  - `int getModifiers()`
    - modifikátory
  - `Class<?> getDeclaringClass()`
    - které třídě atribut patří

java.lang.reflect.Field

- získávání hodnoty atributu
  - `Object get(Object obj)`
    - vrátí hodnotu atributu v objektu obj
    - pokud je atribut primitivního typu, pak je hodnota vrácena v odpovídajícím wrapper typu
  - `boolean getBoolean(Object obj)`
    - vrátí hodnotu boolean atributu v objektu obj
  - `int getInt(Object obj)`
    - vrátí hodnotu int atributu v objektu obj
  - ...
- nastaví hodnoty atributu
  - `void set(Object obj, Object value)`
    - nastaví hodnotu atributu v objektu obj na hodnotu value
  - `void setInt(Object obj, int v)`
  - `void setBoolean(Object obj, boolean b)`
  - ....

java.lang.reflect.Method

- `String getName()`
- `Class<?> getDeclaringClass()`
- `int getModifiers()`
- `Class<?> getReturnType()`
  - návratový typ metody
- `Class<?>[] getExceptionTypes()`
  - pole s typy výjimek, které může metoda vyhodit
- `Class<?>[] getParameterTypes`
  - pole s typy parametrů
  - v pořadí, v jakém jsou deklarovány

java.lang.reflect.Method

- `Object invoke(Object obj, Object... params)`
  - na objektu obj zavolá metodu
  - params – parametry volání metody
    - pokud je metoda bez parametrů, pak params může být null nebo pole délky 0
    - pořadí parametrů v jakém jsou deklarovány
    - hodnoty primitivních typů jsou v příslušné wrapper třídě
  - vrací návratová hodnotu volání metody
    - hodnoty primitivních typů jsou v příslušné wrapper třídě

java.lang.reflect.Constructor<T>

- `String getName()`
- `Class<T> getDeclaringClass()`
- `int getModifiers()`
- `Class<?>[] getExceptionTypes()`
- `Class<?>[] getParameterTypes()`
- `Object newInstance(Object... params)`
  - vytvoří novou instanci třídy
  - pro parametry platí vše jako u metody `invoke()`

java.lang.reflect.Executable

- od Java 8
- `Method` a `Constructor` dědí od `Executable`
- nové metody
  - `public int getParameterCount()`
    - počet parametrů
  - `public Parameter[] getParameters()`
    - parametry
  - ...

- `Parameter`
  - od Java 8
  - `public String getName()`
    - jméno parametru
      - v Java <= 7 nelze jméno parametru získat

java.lang.reflect.Array

- statické metody pro přístup k polím
- `Object newInstance(Class<?> componentType, int length)`
  - vytvoří jednorozměrné pole
- `Object newInstance(Class<?> componentType, int[] dimensions)`
  - vytvoří vícezměrné pole
- `Object get(Object array, int index)`
- `int getInt(Object array, int index)`
- `...`
- `void set(Object array, int index, Object val)`
- `void setInt(Object array, int index, int val)`
- `...`

Reflexe vs. generické typy

- Introspekce probíhá během runtime
  - pozor na *type erasure*

```
class MethodTrouble<T> {
   void lookup(T value) {}
}


Class<?> c = (new MethodTrouble<Integer>()).getClass();
Method m = c.getMethod("lookup", Integer.class);
```

Reflexe vs. generické typy

- `Class` implementuje interface `GenericDeclaration`:
  - `TypeVariable<?>[] getTypeParameters()`
    - vrací seznam generických parametrů deklarovaných danou třídou.
      - lze získat
        - upper bound (***`T extends něco`***)
        - kde je parametr deklarovaný
      - pozor lower-bound (***`T super něco`***) nelze specifikovat u typů!

Anotace

- `Class` implementuje interface `AnnotatedElement`
  - implementován také třídami Field, Method, Package
  - `Annotation[] getAnnotations()`
    - vrací všechny anotace
  - `Annotation[] getDeclaredAnnotations()`
    - vrací všechny anotace, které byly deklarované na dané třídě. Ignoruje zděděné anotace.
  - `<T extends Annotation> T getAnnotation(Class<T> annotationType)`
    - vrací anotaci požadovaného typu (např. Override.class) definovanou na dané třídě a nebo null

Reflexe a moduly

- **`Class<?> Class.forName(Module m, String name)`**

- metoda na Class
  - **`Module getModule()`**

- **`java.lang.Module`**
  - **`String getName()`**
  - **`Set<String> getPackages()`**
  - **`ModuleDescriptor getDescriptor()`**
  - …
  - podrobně později

K čemu je to vše dobré?

- Pluginy
  - Dynamic loading, instantiation
  - Adaptace rozhraní
- Zpracování anotací za běhu
  - viz EJB, Spring, Hibernate
- Patchování/debugování kodu
  - přístup k non-public attributům (viz *Field.setAccesible(true)*),
  - Runtime code generation
- Proxies

java.lang.reflect.Proxy

- vytváření dynamických proxy tříd
  - třída implementující nějaký daný interface a volá metody na nějakém jiném objektu (typicky s nekompatibilním interfacem)
- `static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)`
  - interfaces – pole interfaců, které má proxy implementovat
  - h – objekt zodpovědný za volání metod
- InvocationHandler
  - interface s jednou metodou
  - `Object invoke(Object proxy, Method method, Object[] args)`

java.lang.reflect.Proxy

- příklad

```
InvocationHandler handler = new
                            MyInvocationHandler(...);

Foo f = (Foo)
Proxy.newProxyInstance(Foo.class.getClassLoader(),
                  new Class[] { Foo.class }, handler);
```

Pluginy – příklad

```
interface Plugin {
  void foo();
}

class P1 implements Plugin {
  public void foo() {…}
}
```

Pluginy – příklad (pokrač.)

```
class Main {
  private Plugin[] initPlugins(String[] namesOfPluginsClasses){
    ArrayList<Plugin> ps = new ArrayList<>();
    Class pluginIface = Plugin.class;
    for (String name : namesOfPluginClasses) {
       Class cls = Class.forName(name);
       if (cls.isArray() || cls.isInterface() ||
                  cls.isPrimitive() || ...) {  // report error
         continue;
       }
       if (!pluginIface.isAssignableFrom(cls)) {
         //report error
         continue;
       }
       ps.add(cls.newInstance());
    }
    return ps.toArray(new Plugin [ps.size()]);
  }
...
```