

JAVA

Ještě k funkcionálnímu programování *(pokračování z předchozí přednášky)*

Funkcionální programování




- funkce ve FP ~ „matematická funkce“
 - má parametry
 - vrací výsledek(ky)
 - **nemá vedlejší efekty!!!**
 - **POZOR: I/O operace jsou také vedlejší efekty**
 - nevyhazují výjimky
 - také lze posuzovat jako vedlejší efekt
 - pokud možno jsou líné
- data (seznamy) jsou nemodifikovatelná
 - funkce vrací nová

Líné funkce

- příklad

```
class DebugPrint {  
    private boolean debug;  
    public void setDebug(boolean d) { debug = d; }  
    public void println(String s) {  
        if (debug) { System.out.println(s); }  
    }  
}
```

```
...  
DebugPrint db = new DebugPrint();  
...  
db.println("Name of the user: " + userName);
```

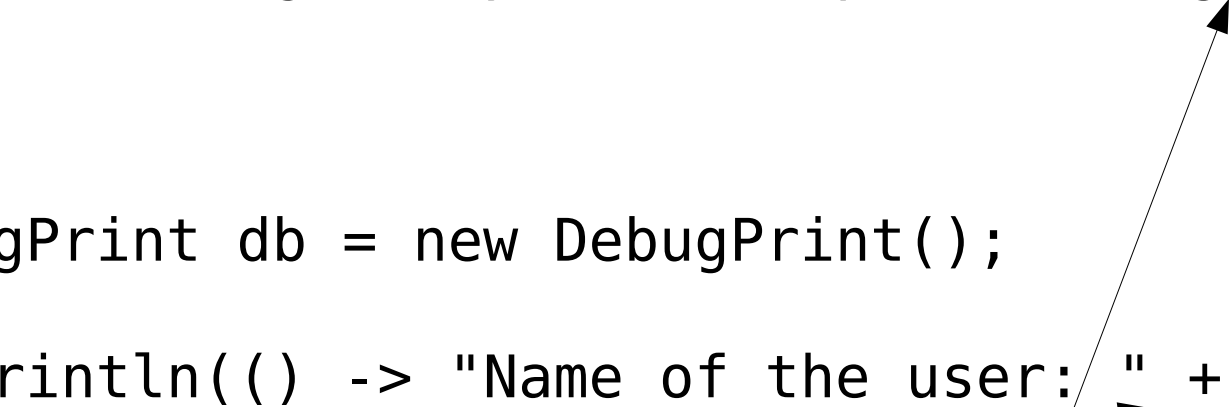


- řetězec je potřeba pouze pokud `debug == true`
ALE vytváří se vždy
 - `new StringBuffer().append(...`

Líné funkce

- lépe

```
class DebugPrint {  
    private boolean debug;  
    public void setDebug(boolean d) { debug = d; }  
    public void println(Supplier<String> c) {  
        if (debug) { System.out.println(c.get()); }  
    }  
}  
  
...  
DebugPrint db = new DebugPrint();  
...  
db.println(() -> "Name of the user: " + userName);
```



- řetězec se vytvoří pouze pokud je potřeba

Nevyhazování výjimek

- při chybě vracet speciální hodnotu
- null není vhodné
 - nelze řetězit volání
- Optional<T>
 - třída
 - kontejner pro hodnotu, které může být null
 - metody
 - boolean isPresent()
 - T get()
 - void ifPresent(Consumer<? super T> consumer)
 - ...
 - získání
 - static <T> Optional<T> empty()
 - static <T> Optional<T> of(T value)
 - static <T> Optional<T> ofNullable(T value)

JAVA

Serializace

Přehled

- "ukládání" celých objektů
 - objekty "přežívají" mezi běhy programu
- persistence
 - lightweight persistence
 - explicitní ukládání a obnovování
- serializované objekty lze přenášet i po síti
- ukládá se stav objektu
 - atributy
- kód třídy objektu musí být dostupný

- potenciálně nebezpečné
 - možná bude v budoucnu odstraněno/nahrazeno

Použití

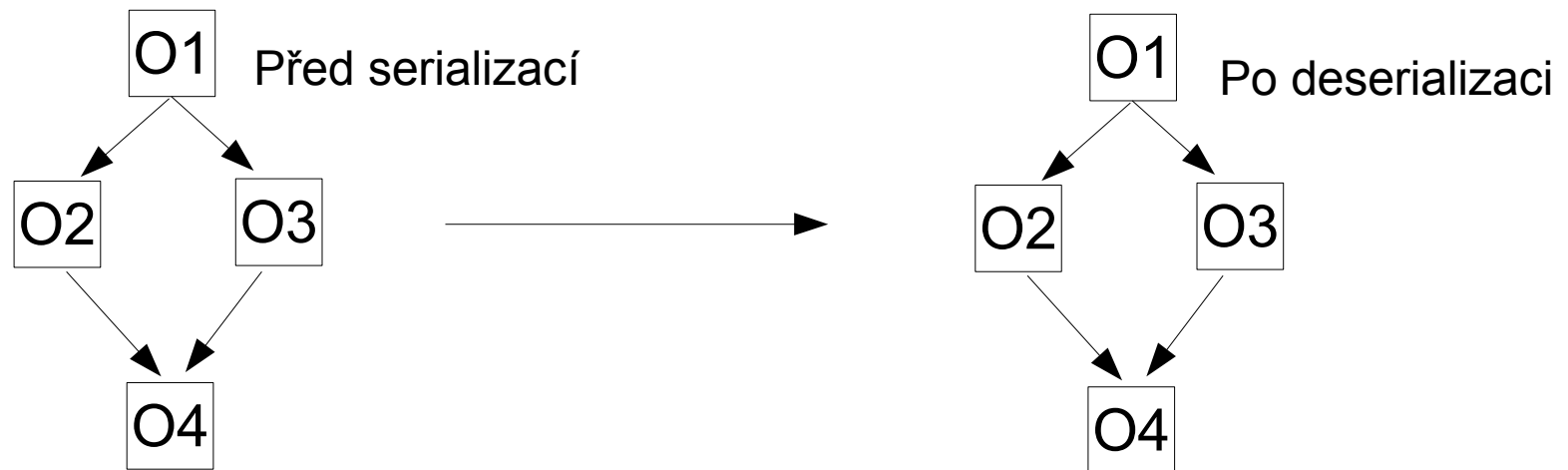
- `java.io.Serializable`
 - prázdný interface
 - serializovatelný objekt ho musí implementovat
- `ObjectOutputStream`
 - potomek `OutputStream`
 - **implementuje** `DataOutput` **a** `ObjectOutput`
 - **metoda** `void writeObject (Object o)`
- `ObjectInputStream`
 - potomek `InputStream`
 - **implementuje** `DataInput` **a** `ObjectInput`
 - **metoda** `Object readObject ()`

Příklad

```
public class Data implements Serializable {
    private int d;
    public Data(int d) {this.d = d;}
    public String toString() {
        return super.toString() + ", d=" + d;
    }
}
...
Data data = new Data(1);
...
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("file.dat"));
out.writeObject(data);
...
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("file.dat"));
data = (Data) in.readObject();
```

Serializace

- serializují/deserializují se všechny atributy (i private)
 - modifikátor atributu `transient`
 - atribut se nebude ukládat
- neukládají se jen primitivní hodnoty ale i reference
 - rekurzivně se ukládají i všechny objekty z atributů objektu
 - při načítání se objekty vytvoří stejně provázané
 - př.



Vlastní serializace

- interface `Externalizable`
 - rozšiřuje `Serializable`
 - dvě metody
 - `void readExternal(ObjectInput in)`
 - `void writeExternal(ObjectOutput out)`
- objekty implementují `Externalizable` místo `Serializable`
- dál je vše stejné (téměř)
- modifikátor `transient` nemá žádný význam
 - ukládání/načítání se provádí přes metody `writeExternal` a `readExternal`
- `writeExternal` a `readExternal` se volají automaticky

Příklad

```
public class Data2 implements Externalizable {
    public Data2() { System.out.println("Data2"); }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Data2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Data2.readExternal");
    }
}
...
Data2 d = new Data2();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data2) o.readObject();
```

Nefunkční příklad

```
public class Data3 implements Externalizable {
    Data3() { System.out.println("Data3"); }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        System.out.println("Data3.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        System.out.println("Data3.readExternal");
    }
}
...
Data3 d = new Data3();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data3) o.readObject(); // nastane výjimka!!
```

Načítání objektů

- implicitní serializace (implementování Serializable)
 - při načítání objektů se nevolá konstruktor
 - objekt se vytváří přímo
- vlastní serializace (implementování Externalizable)
 - nejdřív se zavolá konstruktor
 - základní konstruktor bez parametrů
 - musí být dostupný
 - pak se na objektu zavolá `readExternal()`

Vlastní serializace – jiný postup

- implementovat interface `Serializable`
- přidat 2 „magické“ metody
 - `private void writeObject(ObjectOutputStream stream) throws IOException;`
 - `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`
- obě metody musejí mít přesně dodržet danou hlavičku
 - musejí být `private`
- **v `readObject()` a `writeObject()` lze pomocí metod `defaultReadObject()` a `defaultWriteObject()` vyvolat implicitní uložení/načtení**

Příklad

```
public class Test implements Serializable {
    private String a;
    private transient String b;
    public Test(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    private void writeObject(ObjectOutputStream
stream)
throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
}
    private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String) stream.readObject();
}
}
```


Další „magické“ metody

- `private void readObjectNoData() throws ObjectStreamException`
 - volá se při načítání objektu, pokud některá z jeho tříd (třída a nadtřída) není uložena ve streamu
 - použití – při změně hierarchie mezi uložením a načtením
 - př: uložím objekt třídy Monkey, která dědí od Animal
načítám objekt třídy Monkey, která dědí od Mammal
a ta od Animal (metoda se použije na třídě Mammal)
- *cokoliv* `Object readResolve() throws ObjectStreamException`
 - pokud metoda existuje, deserializace objektu dané třídy vrátí to, co tahle metoda
- *cokoliv* `Object writeReplace() throws ObjectStreamException`
 - pokud existuje, serializuje se to, co vrátí

serialVersionUID

- *cokoliv* static final long serialVersionUID = *hodnota*
 - pokud při deserializaci nesouhlasí uložená hodnota s hodnotou ve třídě, vypadne InvalidClassException
 - nemusí se používat
 - vytvoří se automaticky při používání serializace
 - explicitní deklarace se silně doporučuje

Serializace a std knihovna

- mnoho tříd ve standardních knihovnách implementuje Serializable
- pozor – serializace nemusí fungovat mezi různými verzemi Javy
 - obvykle varování v dokumentaci

Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications ...

JAVA

Preference

Přehled

- balík `java.util.prefs`
- od Java 1.4
- určeno pro ukládání a načítání konfigurace programů
- automatické ukládání/načítání
 - kam se ukládá, záleží na OS
 - zvlášť pro každého uživatele
- pouze primitivní typy a řetězce (max. 8 KB dlouhé)
- dvojice
 - klíč – hodnota
 - neimplementuje interface `Map`
- hierarchická struktura (strom)
 - obvykle jen jeden uzel

Získání

- statické metody na třídě Preferences
- `Preferences userNodeForPackage(Class c)`
 - vrací uzel preferencí asociovaný s balíkem dané třídy
- `Preferences systemNodeForPackage(Class c)`
 - jako předchozí metoda
 - společné pro všechny uživatele
- př:
 - `p = Preferences.userNodeForPackage(Foo.class)`
- jméno uzlu ~ plný název balíku
 - tečky se nahradí lomítky "/"

Příklad

```
public class Prefs {
    public static void main(String[] args) {
        Preferences prefs = Preferences
            .userNodeForPackage(Prefs.class);
        prefs.put("url", "http://somewhere/");
        prefs.putInt("port", 1234);
        prefs.putBoolean("connected", true);
        int port = prefs.getInt("port", 1234);

        String[] keys = prefs.keys();
        for (int i; i<keys.length; i++) {
            System.out.println(keys[i] + ": "+
                prefs.get(keys[i], null));
        }
    }
}
```

Metody

- `String get(String key, String def)`
 - vrací hodnotu klíče
 - musí se zadávat implicitní hodnota
- `int getInt(String key, int def)`
 - jako `get`
 - postupně pro všechny typy
- `void put(String key, String val)`
 - přiřadí klíči hodnotu
 - definována i pro ostatní prim. typy
- `String[] keys()`
 - vrací všechny klíče
- `void flush()`
 - zapíše všechny změny

Metody

- `void clear()`
 - smaže všechny preference v uzlu
- `String name()`
 - jméno uzlu
- `String absolutePath()`
 - absolutní jméno uzlu

- všechny metody jsou bezpečné vůči vláknům
- lze používat i z více JVM naráz

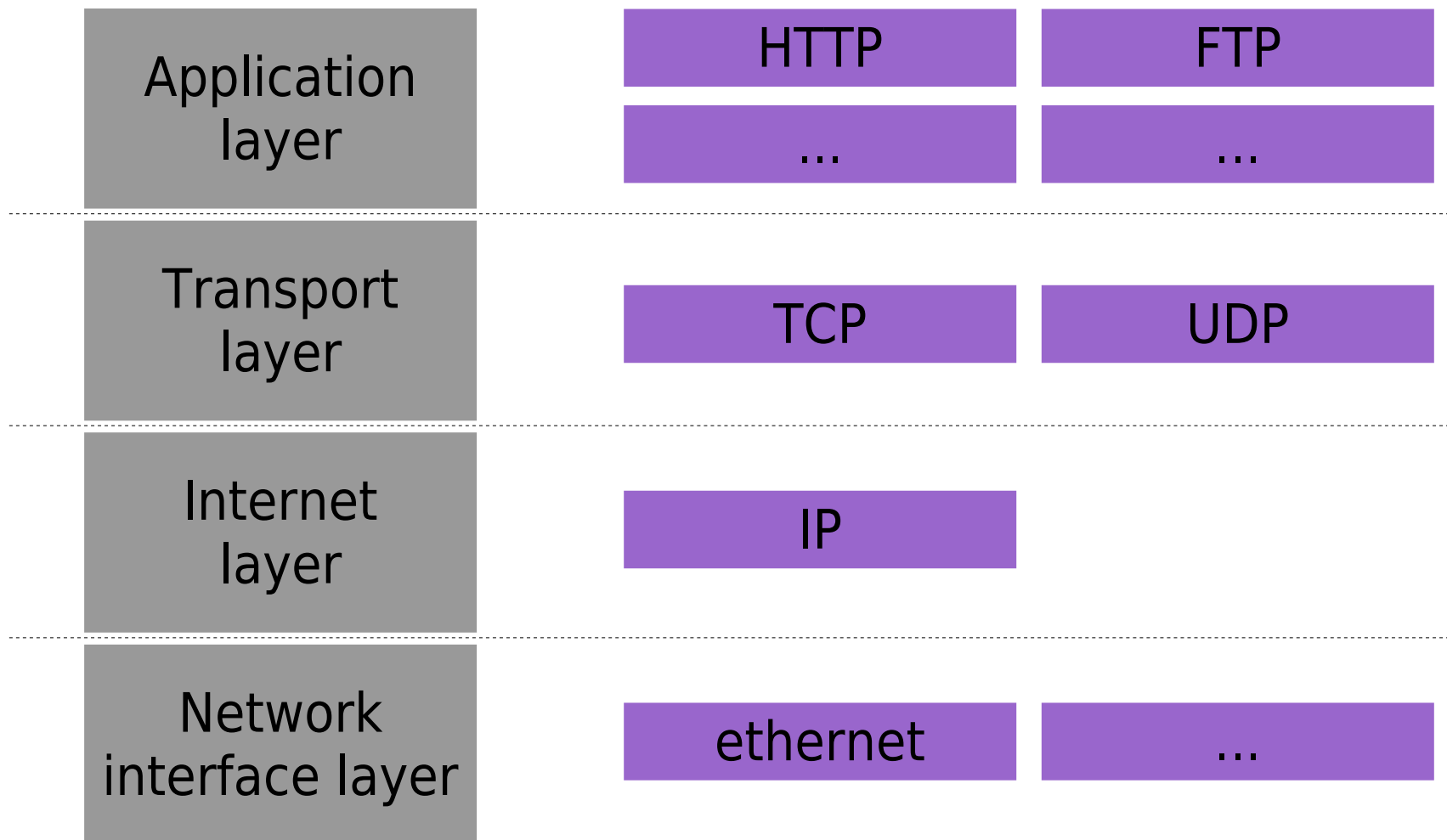
JAVA

Komunikace po síti

Přehled

- balík java.net
- od Java 1.0
- snadná komunikace po síti
- téměř jako používání souborů
 - streamy po síti
- protokoly TCP a UDP
 - Internet

TCP/IP model



URI a URL

java.net.URI

- reprezentace URI
 - unique resource identifier (RFC 2396)
- struktura URI
 - [scheme:]scheme-specific-part[#fragment]
- absolutní URI – obsahuje schema
 - relativní URI – nemá schema
- "opaque" URI – specifická část nezačíná lomítkem
 - př: `mailto:java-net@java.sun.com`
`news:comp.lang.java`
- hierarchické URI – buď absolutní URI začínající lomítkem nebo relativní URI
 - př: `http://java.sun.com/j2se/1.3/`
`../../../../demo/jfc/SwingSet2/src/SwingSet2.java`

java.net.URI

- hierarchické URI - struktura
 - [scheme:][//authority][path][?query][#fragment]
 - authority
 - [user-info@]host[:port]
- všechny části URI jsou typu String, pouze port je typu int
- normalizace URI
 - odstranění a nahrazení "." a ".."

java.net.URI: metody

- `String getScheme()`
- `String getSchemeSpecificPart()`
- `String getPath()`
- `String getHost()`
-
- `boolean isAbsolute()`
- `boolean isOpaque()`
- `void normalize()`
- `URL toURL()`
 - vytvoří URL z URI
 - výjimka pokud nelze

java.net.URL

- URL je speciální případ URI
- unique resource locator
- určování zdrojů na webu
 - `http://www.mff.cuni.cz/`
- podobné metody jako u URI
 - `get...`
- `InputStream openStream()`
 - otevře stream pro čtení souboru určeného pomocí URL
- `URLConnection openConnection()`
 - vytvoří spojení na URL objekt

URLConnection

- reprezentace spojení mezi aplikací a URL
- použití
 1. získání spojení (`openConnection()`)
 2. nastavení parametrů
např. `setUseCaches()`
 3. vytvoření spojení (`connect()`)
vzdálený objekt se stane přístupný
 4. získávání obsahu a informací
obsah – `getContent()`
hlavičky – `getHeaderField()`
streamy – `getInputStream()`, `getOutputStream()`
další – `getContentType()`, `getDate()`, ...

Identifikace (DNS)

InetAddress

- reprezentuje IP adresu
- získání adresy
 - statické metody na InetAddress
 - InetAddress getByName(String host)
 - IP adresa pro dané jméno počítače
 - pro null vrátí localhost
 - InetAddress getByAddress(byte[] addr)
 - IP adresa pro danou adresu
 - délka addr pole – 4 pro IPv4, 16 pro IPv6
 - InetAddress getLocalHost()
 - vrátí adresu pro localhost (127.0.0.1)

Příklad

```
public class InetName {
    public static void main(String[] args) throws
    Exception {
        InetAddress a = InetAddress.getByName(args[0]);
        System.out.println(a);
    }
}
```

```
public class Localhost {
    public static void main(String[] args) throws
    Exception {
        System.out.println(InetAddress.getByName(null));
        System.out.println(InetAddress.getLocalHost());
    }
}
```

Sokety

Přehled

- soket = zakončení spojení
- TCP
 - spolehlivá komunikace
- spojení jsou obousměrná
 - lze získat `InputStream` a `OutputStream`
- třída `ServerSocket`
 - vytvoří "poslouchací" soket
 - metoda `accept()`
 - čeká na příchozí spojení
 - vrátí soket pro komunikaci
- třída `Socket`
 - soket pro komunikaci

Příklad: jednoduchý server

```
try (ServerSocket s = new ServerSocket(6666)) {
    System.out.println("Server ready");
    try (Socket socket = s.accept()) {
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();
        while (true) {
            ...
            in.read();
            ...
            out.write(...);
            ...
        }
    }
}
```


Příklad: jednoduchý klient

```
InetAddress addr = InetAddress.getByName(null);
Socket socket = new Socket(addr, 6666);
try (InputStream in = socket.getInputStream();
     OutputStream out = socket.getOutputStream()) {
    while (...) {
        ...
        out.write(...);
        ...
        in.read();
        ...
    }
}
```

Obsluha příchozích požadavků

- předchozí příklad – jednoduchý server
 - obsluhuje jen jedno spojení
- obsluha více spojení
 - pro každé příchozí spojení vytvořit vlákno

 - nebo

 - channels a třída Selector
 - obsluha více požadavků v jednom vláknu
 - selektor drží množinu soketů
 - metoda `select()` čeká dokud aspoň jeden soket není připraven k použití
 - obdoba funkce `select()` na UNIX systémech

Vícevláknový server

```
class ServeConnection extends Thread {
    private Socket socket; private InputStream in;
    private OutputStream out;
    public ServeConnection(Socket s) throws IOException {
        socket = s; in = ...; out = ...; start();
    }
    public void run() {
        while (true) {
            in.read();
            out.write(...);
        }
    }
}

public class Server {
    public static void main(String[] args) throws
    IOException {
        ServerSocket s = new ServerSocket(6666);
        while(true) {
            Socket socket = s.accept();
            new ServeConnection(socket);
        }
    }
}
```

UDP

Přehled

- nespolehlivá komunikace
- třída DatagramSocket
 - jak pro server tak pro klienta
 - posílají/přijímají se datagramy
 - void send(DatagramPacket d)
 - void receive(DatagramPacket d)
- třída DatagramPacket
 - datagram
 - void setData(byte[] buf)
 - byte[] getData()
 - nastaví nebo vrátí bufer pro datagram
 - int getLength()
 - void setLength(int a)
 - délka dat v datagramu



Verze prezentace J10.cz.2018.01

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