Introduction
About course

- Petr Hnětynka
  - hnetynka@d3s.mff.cuni.cz

- http://d3s.mff.cuni.cz/teaching/vsjava/

- continuation of "Java (NPRG013)"
  - basic knowledge of Java is expected (in the scope of NPRG013)

- 2/2 Zk/Z
Exam/”Započet”

• exam
  - written test
    • as in the winter semester

• “zápočet”
  - home project
    • see the next slide
  - test in the lab
  - 3 homeworks
    • at least 150 points
  - attendance to the practicals
    • > 3 absences => at least 210 points from homeworks
“Zápočet”

• creating a project
  – agreeing a topic till Friday 24\textsuperscript{th} May 2019
    • by email
    • appropriately complex topic
    • non-trivially exploiting a covered technology
  – the project should be submitted till the end of June
    • the latest deadline – Friday 20\textsuperscript{th} September 2019 12:00 (noon)
    • submission by email; only if it is necessary the project is shown personally
Course synopsis

- GUI
- In-depth view of the Java language
  - reflection API
  - generics, annotations
  - classLoaders, security
- Distributed technologies: RMI,...
- Component model JavaBeans
- JEE: Servlets, EJB, Spring,...
- JME: CLDC, MIDP, MEEP
- RTSJ
- Other Java-based technologies: Java APIs for XML, JDBC, JMX,...
- Other languages compiled to Java byte-code
- Android
## Popularity

### Java, summer semester 2019

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Java</td>
<td>15.976%</td>
<td>+0.39%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>12.424%</td>
<td>+0.57%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td>Python</td>
<td>7.574%</td>
<td>+2.41%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td>C++</td>
<td>7.444%</td>
<td>+1.72%</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
<td>Visual Basic .NET</td>
<td>5.903%</td>
<td>+3.02%</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
<td>JavaScript</td>
<td>2.848%</td>
<td>-0.32%</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td></td>
<td>C#</td>
<td>2.846%</td>
<td>-1.61%</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td></td>
<td>PHP</td>
<td>2.771%</td>
<td>-1.12%</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td></td>
<td>SQL</td>
<td>1.900%</td>
<td>-0.46%</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td></td>
<td>Objective-C</td>
<td>1.447%</td>
<td>+0.32%</td>
</tr>
</tbody>
</table>

### Worldwide, Feb 2019 compared to a year ago:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Change</th>
<th>Language</th>
<th>Share</th>
<th>Trend</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>↑</td>
<td>Python</td>
<td>26.42%</td>
<td>+5.2%</td>
</tr>
<tr>
<td>2</td>
<td>↓</td>
<td>Java</td>
<td>21.2%</td>
<td>-1.3%</td>
</tr>
<tr>
<td>3</td>
<td>↑</td>
<td>Javascript</td>
<td>8.21%</td>
<td>-0.9%</td>
</tr>
<tr>
<td>4</td>
<td>↑</td>
<td>C#</td>
<td>7.57%</td>
<td>-0.5%</td>
</tr>
<tr>
<td>5</td>
<td>↓↓</td>
<td>PHP</td>
<td>7.34%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>C/C++</td>
<td>6.56%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>R</td>
<td>1.13%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Objective-C</td>
<td>3.04%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Swift</td>
<td>2.56%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Matlab</td>
<td>1.98%</td>
<td>-0.4%</td>
</tr>
</tbody>
</table>

### Language Rank

1. Python | 100.0
2. C++    | 99.7
3. Java   | 97.5
4. C      | 96.7
5. C#     | 89.4
6. PHP    | 84.9
7. R      | 82.9
8. JavaScript | 82.6
9. Go     | 76.4
10. Assembly | 74.1
Popularity

TIOBE Programming Community Index

Source: www.tiobe.com

source: http://www.tiobe.com/tiobe_index
Java
Type system

- strongly typed language
  - classes
  - primitive types (int, boolean, char,...)
- "everything" is in a class
- no global variables, functions,...
  - static methods and fields can seen as global elements
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);
    }
}

The program prints out:
  a) 2  4
  b) 1  3
  c) 3  5
  d) 4  4
  e) cannot be compiled
Solution

• correct is  a)  2  4

• { ...... } in the class body
  – initializer
  – executed when an instance is created
  – used for initialization of anonymous inner classes

• static { ...... }
  – static initializer
  – executed during class loading to VM
  – can access only static elements of the class
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);
    }
}

Results:
  a) 1   3   5
  b) 2   3   5
  c) 2   4   5
Solution of test 2

- correct us  C)  2  4  5

- the initializer is executed before execution of a constructor

- first, a superclass is initialized
  - initializers and constructors
Exceptions and initializers

- initializers can throw only exceptions that are defined in constructors
  - there must be at least one constructor

- initializers of anonymous inner classes can throw any exceptions
  - the class is instantiated just once
  - no problem to catch/declare the exceptions
Static initializers

- have to terminate without an exception
  - otherwise cannot be compiled
- run in the order as in the source file
- cannot contain `return`
  - otherwise cannot be compiled
Visibility in classes

• is it possible to change element's visibility in children?
  – e.g.
    ```java
class A { public void foo() {} }
class B extends A { private void foo() {} }
```

• visibility cannot be “restricted” but can be “increased”

• why
  – if it would be possible
    ```java
class A { public void foo() {} }
class B extends A { private void foo() {} }
```
  – then the following code would be possible
    ```java
    A a = new B();
a.foo();
```
Type changes

- covariant change – from specific to generic
- contravariant – vice versa

- arrays in Java are covariant
  
  ```java
  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;
  ```

- what would happen if we try this?
  ```java
  myNumbers[0] = 3.14;
  ```
Covariance

- `myNumber[0] = 3.14;`
  - can be compiled
  - exception at runtime
Reflection API
Overview

- Reflection
  - changes structure/state of objects

- Introspection
  - exploring a structure of objects

- allows
  - obtaining information about class, fields, methods
  - creating instances
  - calling methods
  - ...

- **package** `java.lang.reflect`
- **class** `java.lang.Class`
java.lang.Class<T>

- an instance of the **Class** class represents a class or interface in a running program
- primitive types are also represented as instance of the **Class** class
- it has no constructor
- instances created automatically during loading the class to JVM
  - classes are loaded to JVM just before their first usage
- since Java 5 – a generic type
  - T – the type of the class represented by this Class object
    - ex.: for `String ~ Class<String>`
    - if unknown, then `Class<?>`
obtaining instances of `Class`

- `getClass()`
  - method of the `Object` class
  - returns the class of the object, on which it is called
- *literal `class`* (it is an *expression* of the type `Class`)
  - `JmenoTridy.class`
  - the class for the given type
- `Class.forName(String className)`
  - static method
  - returns the class with the given name
- for primitive types
  - static filed `TYPE` of the wrapper classes
    - `Integer.TYPE`
  - *literal `class`
    - `int.class`
java.lang.Class<T>

• the type after obtaining the instance

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

Class<String> clazz2 = String.class;

Class<Integer> clazz3 = int.class;

- but

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

• classes are loaded to JVM by a classloader
  – java.lang.ClassLoader
  – the standard classloader looks up classes in CLASSPATH
  – it is possible to create own classloader
  – Class.forName(String className, boolean initialize, ClassLoader cl)
    • loads a class with the given classloader and returns the instance of the class
  – getClassLoader()
    • a method of the Class class
    • returns a classloader, which loaded the class
      – *warning*: the type of an object is represented not only by the Class but also by a classloader that loaded the given class
        • in detail will be later
java.lang.Class<T>: methods

- boolean isPrimitive()
- boolean isArray()
- boolean isInterface()
- boolean isEnum()
- boolean isAnnotation()
  - tests whether the class represents a primitive type resp. array resp. interface resp. enum resp. annotation
- boolean isInstance(Object o)
  - tests whether the given object is an instance of the class
  - equivalent to the instanceof operator
- boolean isAssignableFrom(Class<?> cls)
  - tests whether the class/interface is the same as cls or a superclass/superinterface of cls
  - i.e. whether an object of the type cls can be assigned to a variable of the type, on which the method is called
java.lang.Class<T>: methods

- `String getName()`
  - returns name of the class (interface,...)
  - for primitive types returns their names
  - for an array returns a string beginning with with `[ chars (as much as the array has dimensions) and then identification of the element type
Z..boolean, B..byte, C..char, D..double, F..float, I..int, J..long, S..short, Lclassname..class or interface

```java
String.class.getName()    // returns "java.lang.String"
byte.class.getName()      // returns "byte"
(new Object[3]).getClass().getName()  // returns "[Ljava.lang.Object;"
(new int[3][4][5][6][7][8][9]).getClass().getName() // returns "[[[[[[[I"
```
java.lang.Class<T>: methods

- **Package getPackage()**
  - returns the package in which the class is defined
  - java.lang.Package
    - information about the package
- **Class<? super T> getSuperclass()**
  - returns the super class
  - returns null for the Object class, primitive types and interfaces
- **Class<?>[] getInterfaces()**
  - returns all implemented interfaces
  - if the class does not implement any interface, it returns an array with 0 elements
  - for primitive types it also an array with 0 elements
java.lang.Class<T>: methods

- Method[] getMethods()
  - returns all methods of the class (public)
- Field[] getFields()
  - returns all fields of the class (public)
- Constructor<?>[] getConstructors()
  - returns all constructors of the class (public)
- Method[] getDeclaredMethods()
- Field[] getDeclaredFields()
- Constructor<?>[] getDeclaredConstructors()
  - returns all declared methods/fields/constructors of the class
  - it does not return inherited elements
java.lang.Class<T>: methods

- Field getField(String name)
- Field getDeclaredField(String name)
  - returns a field of the given name
- Method getMethod(String name, Class<?>... paramTypes)
- Method getDeclaredMethod(String name, Class<?>... paramTypes)
  - returns a method of the given name and given types of parameters
- Constructor<T> getConstructor(Class<?>... paramTypes)
- Constructor<T> getDeclaredConstructor(Class<?>... paramTypes)
  - returns a constructor of the given types of parameters
java.lang.Class<T>: methods

- Class<?> getDeclaringClass()  
  - returns a class or interface in which the class/interface is declared  
  - for inner classes
- Class<?>[] getClasses()  
  - returns all classes/interfaces declared in the class or superclasses
- Class<?>[] getDeclaredClasses()  
  - returns all classes/interfaces declared in the class
- Class<?> getComponentType()  
  - returns a type of the array elements  
  - for non-arrays, it returns null
public URL getResource(String name)
public InputStream getResourceAsStream(String name)
- reads a resource
  - images, ...., anything
- data are loaded by a classloader ==> loading by the same rules as loading classes
- a name of the resource ~ hierarchical name as for classes
  - dots are replaced with slashes '/'
T cast(Object o)
- since Java 5
- in <=JDK 1.4 would have no meaning
T[] getEnumConstants()
- returns an array with values of the enum
  - if the class does not represent an enum, it returns null
java.lang.Class<T>: instance

- `T newInstance()`
  - creates a new instance of the class
  - a parameter-less constructor is used
    - it is the same as usage of new AClass()
  - deprecated since Java 9
    - use `getDeclaredConstructor().newInstance()`

- creating new instance of the class using different constructors
  - the class `java.lang.reflect.Constructor<T>`
Modifiers

• `int getModifiers()`
  - method of `java.lang.Class`
  - returns modifiers encoded in an integer

• `java.lang.reflect.Modifiers`
  - decoding the integer with modifiers
  - static methods
    • `boolean isPublic(int mod)`
    • `boolean isStatic(int mod)`
    • `boolean isSynchronized(int mod)`
    • ...
    • `void toString(int mod)`
      - returns a readable string with modifiers
java.lang.reflect.Field

- information about fields
- accessing fields

- methods
  - String getName()
    - name of the fields
  - Class<?> getType()
    - type
  - int getModifiers()
    - modifiers
  - Class<?> getDeclaringClass()
    - in which class it is declared
java.lang.reflect.Field

• obtaining value of the field
  - Object get(Object obj)
    • returns a value of the field of the object obj
    • for primitive type fields, the value is returned in the corresponding wrapper type
  - boolean getBoolean(Object obj)
    • returns value of the boolean field in the object obj
  - int getInt(Object obj)
    • returns value of the int field in the object obj
  - ...

• setting value of the field
  - void set(Object obj, Object value)
    • sets the value to the field in the object obj
  - void setInt(Object obj, int v)
  - void setBoolean(Object obj, boolean b)
  - ....
java.lang.reflect.Method

- String getName()
- Class getDeclaringClass()
- int getModifiers()
- Class<?> getReturnType()
  - returning type of the method
- Class<?>[] getExceptionTypes()
  - array of types of exceptions the method can throw
- Class<?>[] getParameterTypes
  - returns an array with types of the parameters
  - in the declared order
Object invoke(Object obj, Object... params)

- calls the method on the object obj
- params – values for parameters for the call
  - for methods with no parameters, the params can be null or zero-length array
  - parameters in the declared order
  - values of primitive types in the corresponding wrapper
- returns returning value of the method call
  - values of primitive types in the corresponding wrapper
java.lang.reflect.Constructor<T>

- String getName()
- Class<T> getDeclaringClass()
- int getModifiers()
- Class<?>[] getExceptionTypes()
- Class<?>[] getParameterTypes()
- Object newInstance(Object... params)
  - creates new instance of the class
  - the same rules for the params like for the method invoke()
java.lang.reflect.Executable

- since Java 8
- Method and Constructor extends Executable
- new methods
  - public int getParameterCount()  
    - number of formal parameters  
  - public Parameter[] getParameters()  
    - parameters  
  - ...

- Parameter
  - since Java 8
  - public String getName()  
    - name of the parameter  
      - in Java <= 7 the parameter's name cannot be obtained
java.lang.reflect.Array

• static methods for accessing arrays
• Object newInstance(Class<?>
  componentType, int length)
  – creates a single-dimension array
• Object newInstance(Class<?>
  componentType, int[] dimensions)
  – creates a multi-dimension array
• 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)
Reflection vs. generics

- Introspection is performed at runtime
  - be careful with the type erasure

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

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

- Class implements the interface GenericDeclaration:
  - TypeVariable[][] getTypeParameters()
    - returns an array with generic parameters declared by the class
      - is it possible to obtain
        - upper bound (T extends something)
        - declaring item
      - warning – lower-bound (T super something) cannot be specified for types!
Annotations

- Class implements the AnnotatedElement interface
  - which is also implemented by the classes Field, Method, Package

- Annotation[] getAnnotations()
  - returns all annotations
- Annotation[] getDeclaredAnnotations()
  - returns all annotations declared on the given class; it ignores inherited annotations
- <T extends Annotation> T getAnnotation(Class<T> annotationType)
  - returns an annotation of the given type (e.g. Override.class) declared on the class or null
Reflection and modules

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

- method on Class
  - Module getModule()

- java.lang.Module
  - String getName()
  - Set<String> getPackages()
  - ModuleDescriptor getDescriptor()
  - ...
  - in more detail later
What can be done with it?

- Plugins
  - Dynamic loading, instantiation
  - Interface adaptation

- Processing annotations at runtime
  - see EJB, Spring, Hibernate

- Patching/debugging code
  - accessing non-public fields (see `Field.setAccessible(true)`),
  - Runtime code generation

- Proxies
java.lang.reflect.Proxy

• creating dynamic proxy classes
  – a class implementing a given interface and it calls methods from a different object (typically with an incompatible interface)
• static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
  – interfaces – an array of interfaces to be implemented by the proxy
  – h – an object responsible for calling the methods
• InvocationHandler
  – the interface with the single method
  – Object invoke(Object proxy, Method method, Object[] args)
java.lang.reflect.Proxy

• example

    InvocationHandler handler = new
        MyInvocationHandler(...);

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

interface Plugin {
    void foo();
}

class P1 implements Plugin {
    public void foo() {
    }
}
Plugins – example (cont.)

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()]);
    }
    ...
}