JAVA

Note about the Reflection API

Overview

- reflection, introspection
- allows for
  - obtaining information about classes, fields, methods
  - creating objects
  - calling methods
  - ...
- the package `java.lang.reflect`
- the class `java.lang.Class<T>`

java.lang.Class

- an instance of the class **Class** represents a class (interface, enum,...) in a running program
- primitive types also represented as instances of **Class**
- no constructor
- instances created automatically during loading the class code to JVM
  - classes are loaded to JVM when firstly used

java.lang.Class

- obtaining an instance of **Class**
  - `getClass()`
    - the method of the Object class
    - returns the class of the object on which was called
  - the class literal
    - `JmenoTridy.class`
    - the class for the given type
  - `Class.forName(String className)`
    - static method
    - returns the class of the given name
  - for primitive types
    - the static attribute TYPE on the wrapper classes
      - `Integer.TYPE`
    - the literal `class`
      - `int.class`

java.lang.Class

- class are loaded to JVM by a *classloader*
  - `java.lang.ClassLoader`
  - the standard classloader looks up classes in CLASSPATH
  - own classloaders can be created
  - `Class.forName(String className, boolean initialize, ClassLoader cl)`
    - loads the class by the given classloader and returns an instance of the Class
  - `getClassLoader()`
    - the method of Class
    - the classloader, which loaded the class

java.lang.Class: methods

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

```
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: methods

- `public URL getResource(String name)`
- `public InputStream getResourceAsStream(String name)`
  - reads a resource
    - image, ..., anything
  - data loaded by a classloader => loading by the same rules as loading classes
  - a name of the resource ~ a hierarchical name as of classes
    - dots replaced by `'/'`

java.lang.Class: methods

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

Usage of Reflection API

- information about code
- dynamic loading
- plugins
- proxy classes
- …

- more details in summer semester

jar

Overview

- creating archives composed of .class files
- JAR ~ Java Archive
- file
  - extension .jar
  - format – ZIP
  - file META-INF/MANIFEST.MF
    - description of the content
- usage – distribution of software
  - CLASSPATH can contain .jar files
  - .jar files can be directly executed
- can contain also other files than .class files
  - images
  - audio
  - anything else

Usage

- creating an archive

  ```
  jar cf file.jar *.class
  ```

  - creates the file.jar with all .class files
  - adds the MANIFEST.MF file to it

  ```
  jar cmf manifest file.jar *.class
  ```

  - creates the file.jar with the given MANIFEST file

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

  - no compression

  - see documentation for other parameters

- API for working with jar files
  - java.util.jar, java.util.zip

MANIFEST.MF file

- list of tuples
  - name : value
  - inspired by the standard RFC822
- tuples can be grouped
  - groups separated by an empty line
  - main group (the first one)
  - groups for individual entries in the archive
- length of lines – max 65535
- end of lines
  - CR LF, LF, CR

MANIFEST.MF files

- main group
  - Manifest-Version
  - Created-By
  - Signature-Version
  - Class-Path
  - Main-Class
    - applications can be launched

    java -jar archive.jar
- other section
  - the first tuple

    Name: path_to_the_entry_in_the_archive

Jar and Ant

- the task **jar**
  - parameters
    - destfile, basedir, includes, excludes, manifest
  - inner elements
    - manifest
  - example

```
<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

- similar to java.util.zip

- `JarInputStream`, `JarOutputStream`
  - children of ZipInputStream and ZipOutputStream
  - JarInputStream has the getManifest() method
- `JarEntry`
  - child of `ZipEntry`
  - obtaining attributes
- `Manifest`
  - the MANIFEST.MF file

Modules

Modules

- a module
  - explicitely defines what is provided but also what is ***required***

- why?
  - the *classpath* concept is “fragile”
  - no encapsulation

Modular apps – motivation

- why
  - applications get more complex
  - assembled from pieces
  - developed by distributed teams
  - complex dependencies
  - good architecture
    - know your dependencies
    - manage your dependencies

Module declaration

- module-info.java

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

- modular artifact
  - modular JAR – JAR with module-info.class
  - a new format JMOD
    - a ZIP with classes, native code, configuration,...

Modules and JDK

- JDK std library modularized too
  - java.base – always „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

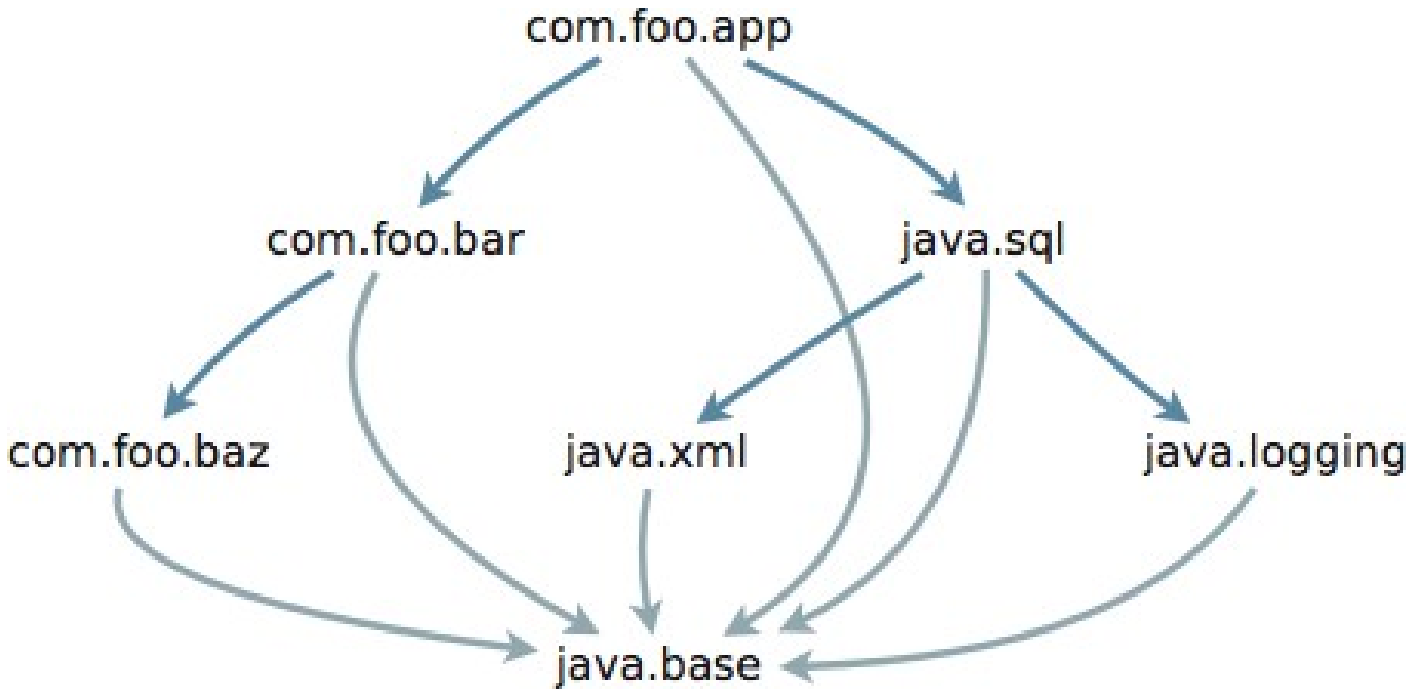
- When one module depends directly upon another

  Module ***reads*** another module (or, equivalently, second module is ***readable*** by first)

- ***Module path*** – equivalent to classpath
  - but for modules
    - -p, --module-path

Module graph

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



Compatibility with “old” Java

- Classpath still supported
  - in fact – modules are “optional”
- Unnamed module
  - artefacts outside any module
    - “old” code
  - reads every other module
  - exports all of its packages to every other module

Modules

- more details in summer semester

JAVA

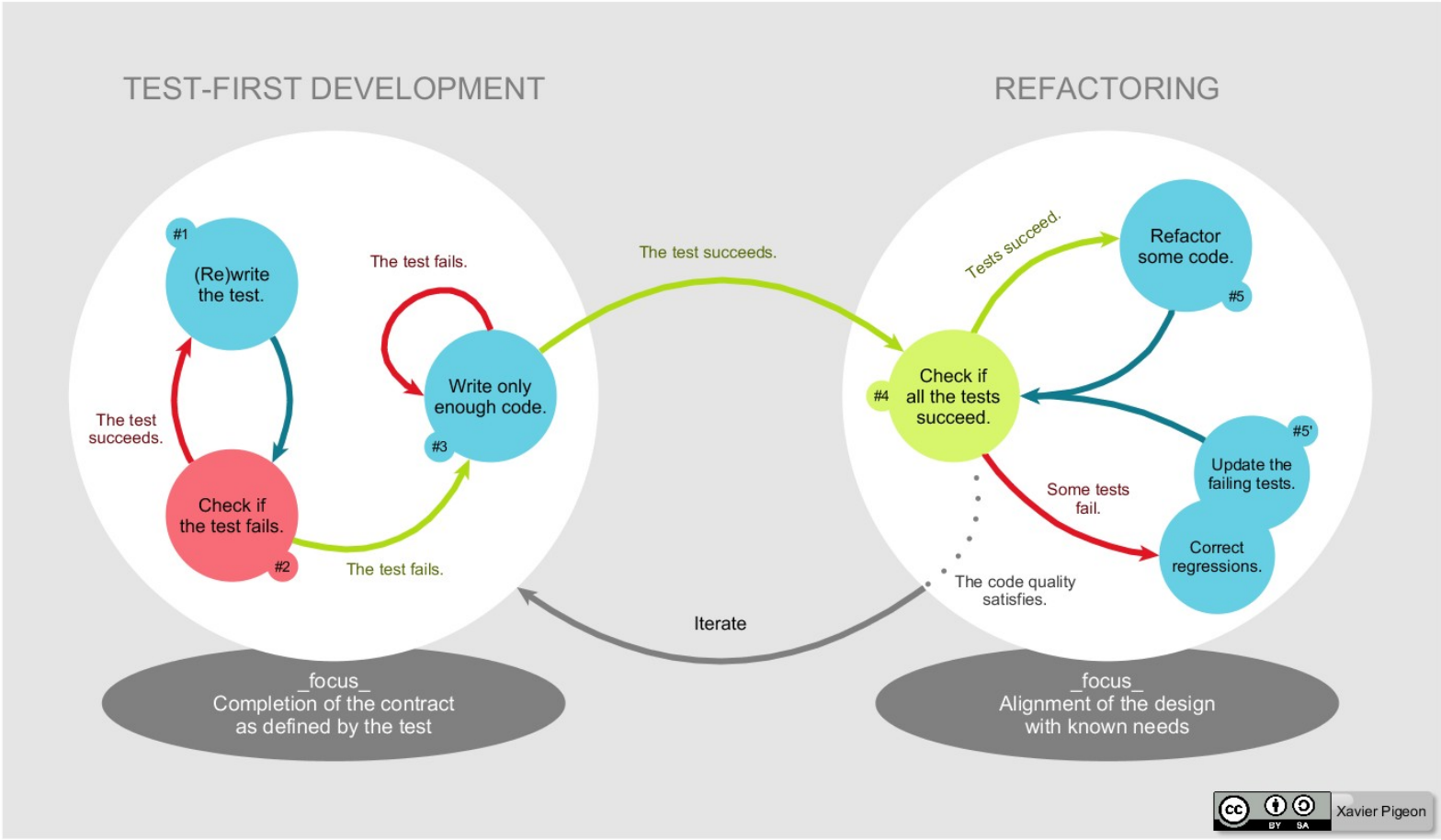
Unit testing

Introduction

- unit testing
  - testing “small” units of functionality
  - a unit – independent on other ones
    - tests are separated
    - creating helper objects for tests
      - context
  - typically in OO languages
    - unit ~ method
  - ideally – unit tests for all units in a program
    - typically in OO languages
      - for all public methods

Test-driven development

- tests first



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

JUnit

- support for unit testing in Java
- http://www.junit.org/
- usage based on annotations
  - older versions based on inheritance and naming conventions
- slightly different usage in different versions
  - 5, 4, 3

Usage

- test methods marked by the @Test annotation
- JUnit is run on a set of classes
  - searches in them all @Test methods
  - executes them

- other annotations
  - @BeforeEach (@Before)
    - a method run before each test
    - intended for “environment” preparation
  - @AfterEach (@After)
    - a method run after each test
    - intended for “cleaning”
  - @BeforeAll (@BeforeClass)
    - a method run before all tests in the given class
  - @AfterAll (@AfterClass)
    - a method run after all tests in the given class

Example

```
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...
  - static methods of org.junit.jupiter.api.Assertions (org.junit.Assert)
  - testing conditions in tests
  - test fails if assert... fails
    - assert...() throws AssertionError

- in general
  - test is successful if the method terminates regularly
  - test fails if the method throws an exception

Testing exceptions

- how to test “correctly” thrown exceptions?

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

- in older versions

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

Running tests

- from code

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

- from command line

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

- from Ant
  - the task junit

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

- from Maven
  - mvn test
- from IDE

TestNG

- http://testng.org/
- inspired by JUnit
- slightly different set of features
  - originally
  - now, more-or-less the same
- basic usage is the same

Java

Reactive programming

Reactive programming (RP)

- data streams and propagating of changes in a program
  - data changes are automatically propagated

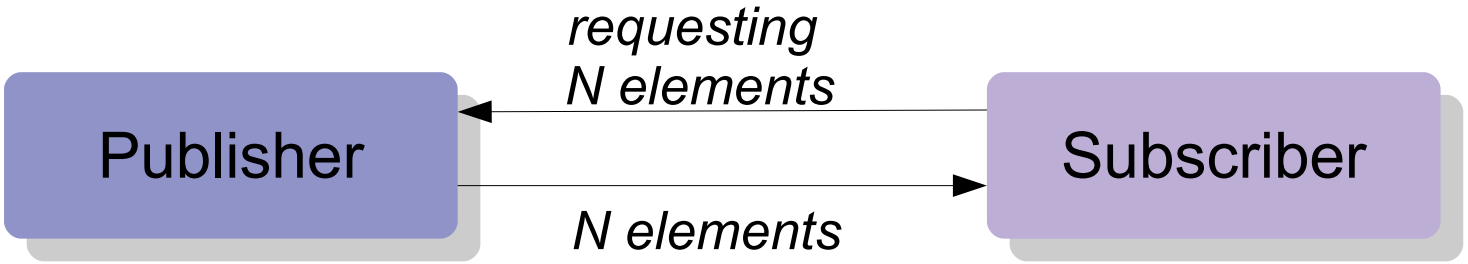
- publisher-subscriber
  - architectural pattern
  - one of particular models for RP

  - publisher publishes data
  - subscriber asynchronously data consumes
  - there can be processor between P and S transforming data

- why RP
  - simpler code, more efficient, …
  - "an extension" of the stream API

Publisher-Subsriber in Java

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



- „a combination of iterator and observer patterns“

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
  - implements the Publisher interface
  - asynchronously publishes given data
  - the constructor without parameters
    - uses ForkJoinPool.commonPool()
  - other constructors – an argument for an executor
  - methods
    - subscribe(Flow.Subscriber<? super T> subscriber)
    - submit(T item)
    - ...

Observer pattern

- an object (observer) „observes“ another object (observable) – if the other object changes, it notifies all its observers
  - java.util.Observer
  - java.util.Observable
    - warning – Deprecated since Java 9 (replaced by Flow)

- usage
  - UI
    - Observable – UI components
    - Observer – reactions to UI events

Java

More about threads

ThreadLocal

- own copy for each thread
- typically used as static fields

- methods

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

Java

What next...

What next

- **NPRG021 Advanced programming for Java platform**
  - summer 2/2
  - synopsis
    - GUI (Swing, JavaFX)
    - Modules, Reflection API, Classloaders, Security
    - Generics, annotations
    - RMI
    - JavaBeans
    - Java Enterprise Edition: EJB, Servlets, Java Server Pages, Spring,...
    - Java Micro Edition: Java for mobile and embedded systems, CLDC, MIDP, MEEP
    - RTSJ, Java APIs for XML, JDBC, JMX,…
    - Kotlin and other “Java” languages
    - Android
  - partially mandatory for NPRG059 Advanced Programming Praxis
    - a mandatory course for several Master study branches