Few words about how to write [disputably] nice code

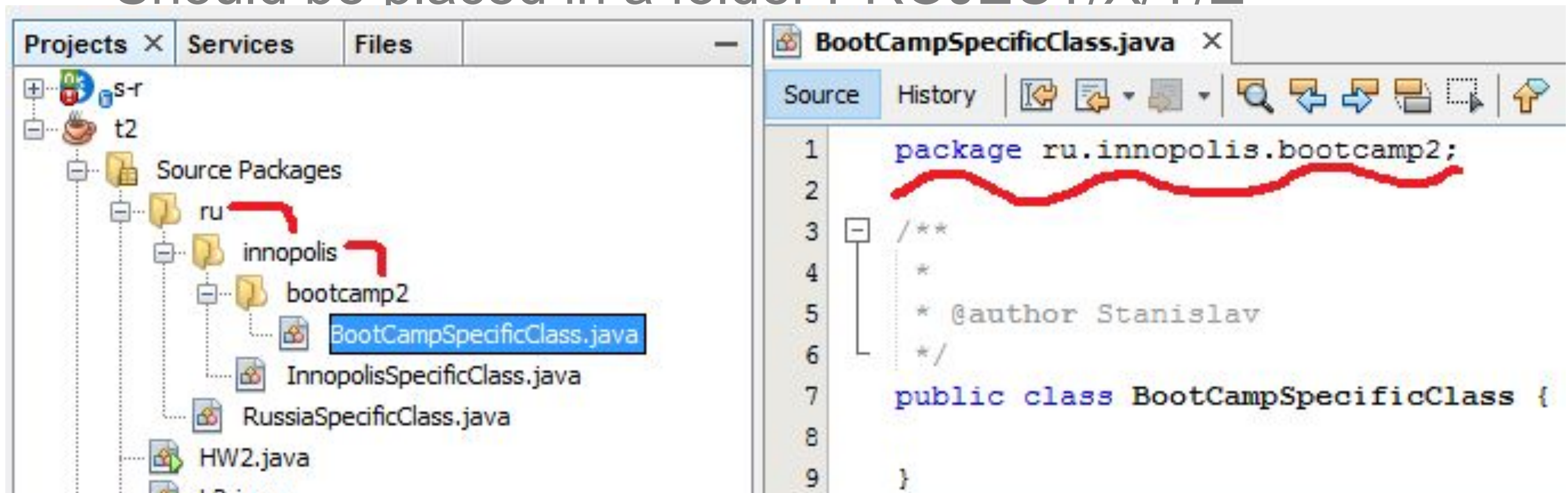
Packages

Package system - way of hierarchical organization of the project code

In Java packages map to file system.

Each file belonging to a package X.Y.Z:

Should be placed in a folder PROJECT/X/Y/Z



Package visibility

```
package ru.innopolis.bootcamp2;
public class PublicClass { }
```

```
package ru.innopolis.bootcamp2;
class DefaultClass { }
```

```
1   package ru.innopolis;
2
    import ru.innopolis.bootcamp2.DefaultClass;
4   import ru.innopolis.bootcamp2.PublicClass;
```

Package naming conventions

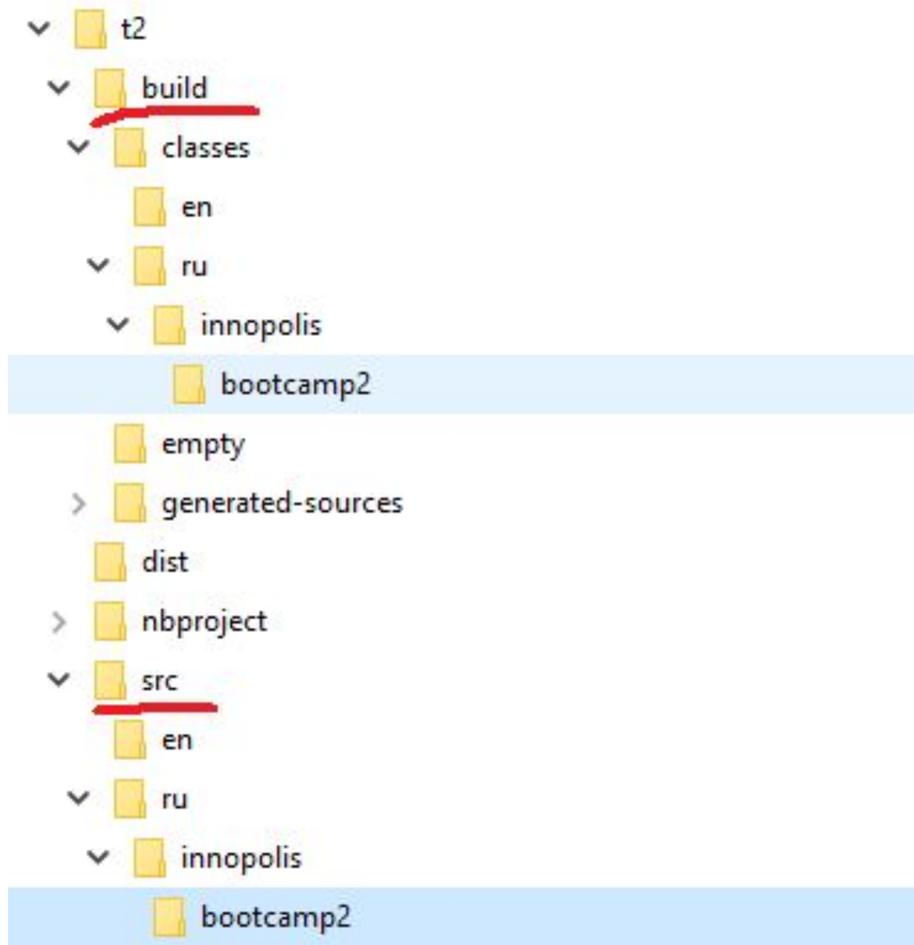
A name for a Java package must be a sequence of one or more valid Java identifiers separated by dots (“.”)

```
package java.lang;

package java.io;

package java.awt;

package ru.innopolis.iis.mlkr;
```

Usually, the letters in the name of a package are all lowercase

If a package is to be widely distributed, it is a common convention to prefix its name with the **reverse Internet**

Packages in file system



BootCampSpecificClass.java
DefaultClass.java
PublicClass.java

Build in packages

We build and run with respect of the package

One class

```
javac ru/innopolis/bootcamp2/BootCampSpecificClass.java
```

```
cd ru/innopolis/bootcamp2
  javac BootCampSpecificClass.java
```

```
java ru/innopolis/bootcamp2/BootCampSpecificClass
```

~~`java DefaultClass`~~

All classes in package

```
javac ru/innopolis/bootcamp2/*.java
```

All classes recursively (build tree)

Ant, Maven

```
# Linux
$ find -name "*.java" > sources.txt
$ javac @sources.txt

:: Windows
> dir /s /B *.java > sources.txt
> javac @sources.txt
```

Referencing the package

```
1    package en;
2
3    import java.io.BufferedReader;  // single class
4    import java.util.*;             // all classes at this level
5    import ru.innopolis.bootcamp2.*;
6
7    public class Foreigner {
8        BufferedReader reader;
9        Scanner scanner;            // we got him from java.util
10       PublicClass pc;
11       java.io.Console console;    // explicitly
12   }
```

Stating coding standards

Coding standards - usually internal corporate document helping to organize the code

- [Start from Java Code Conventions]
- Keep it simple (less than 20 rules)
- Better if developed by the team
- Be not too specific
- Avoid bad standards
- Evolve it over time

Stating coding standards

7.4 if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

Note: `if` statements always use braces, `{}`. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

Common rules

Each Java source file (.java file) has the following structure:

- Introductory comments
- Declaration of package (if needed)
- Import instruction (if needed)
- Definition of classes and interfaces (in Java you can store single class* in a file)

What is comment

Comment - is a piece of text that do not affect compilation

Single line:

```
// something will happen here
int x = 4;    //TODO: implement calculation of x!
```

Multiple lines (inline):

```
obj.callSomeMethod(a /* very important param */, b, c);
/* this is
Very long
Multiline
Comment */
```

Special comments

Use “**//XXX**” in a comment to flag something that is bogus but works

Use “**//FIXME**” to flag something that is bogus and broken

Use “**//TODO**” to flag something that should be implemented

Special comments

```java
//FIXME: this method should return double!
public static int lengthOfCircle(double radius) {
    double PI = 4.0; //XXX: no idea, but works with this value
    int length = 0;  //TODO: i don't remember the formula
    return length;
}
```

Introductory comment (javadoc)

```
/**
 * @deprecated if you recommend not to use a class
 *             to preserve backward compatibility
 *
 * @see OtherClass
 *
 * @serial SERIAL_NUMBER
 *
 * @since WHICH.VERSION.OF.THE.PROJECT/LIBRARY
 *
 * @version 1.0.0.1
 *
 * @author Stanislav Protasov
 */
```

*Javadoc support **html**

http://www.codenet.ru/webmast/java/JavaDoc/

Introductory comment (javadoc)

Tag & Parameter	Usage
@author *John Smith*	Describes an author.
@version *version*	Provides software version entry. Max one per Class or Interface.
@since *since-text*	Describes when this functionality has first existed.
@see *reference*	Provides a link to other element of documentation.
@param *name description*	Describes a method parameter.
@return *description*	Describes the return value.
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.
@deprecated *description*	Describes an outdated method.
{**@inheritDoc**}	Copies the description from the overridden method.
{**@link** *reference*}	Link to other symbol.
{**@value** *#STATIC_FIELD*}	Return the value of a static field.
{**@code** *literal*}	Formats literal text in the code font. It is equivalent to <code>{@literal}</code>.
{**@literal** *literal*}	Denotes literal text. The enclosed text is interpreted as not containing HTML markup or nested javadoc tags.

Introductory comment

```
/*
* @(#)Blah.java 1.82 99/03/18
*
* Copyright (c) 1994-1999 Sun Microsystems, Inc.
* 901 San Antonio Road, Palo Alto, California,
* 94303, U.S.A. All rights reserved.
*
* This software is the confidential and
* proprietary information of Sun Microsystems,
* Inc. ("Confidential Information"). You shall
* not disclose such Confidential Information and
* shall use it only in accordance with the terms
* of the license agreement you entered into
* with Sun.
*/
```

Class/Interface arrangement

Instance variables

Constructors

Methods

```
public class DefaultClass {

    private static final int VERSION = 42;
    protected static String name;

    private String hash;

    public DefaultClass() {
        //some code
    }

    public DefaultClass(String hash) {
        this.hash = hash;
    }

    public static void main(String[] args) {

    }
}
```

http://developer.alexanderklimov.ru/android/java/constructor.php

Class/Interface arrangement

	Class	Package	Subclass	World
public	+	+	+	+
protected	+	+	+	
no modifier	+	+		
private	+			

+ : accessible
  : not accessible

Naming conventions

Divided into

- Conventions about how to give **meaningful** names
- Conventions about how names must be **written**

All of the names in your program should convey information about the purpose of the item they refer

Use not abbreviations for your names (only if they are in common use in normal speech)

Use descriptively named variables

Avoid ambiguous words

Class Names (same for interfaces)

Should be **singular nouns**, referring to the object they represent

First letter and each internal word of class or interface names is capitalized (UpperCamelCase)

```
Train, Event, Station
```

Do not put hierarchy information in class names, unless real-world names bear this information

```
EmployeePerson, SecretaryEmployeePerson
```

Method names

Verbs or verbal forms in mixed cases

Starting with lower letter and each internal word should be capitalized (lowerCamelCase)

Methods returning a boolean usually named with verb phrases expressing the question

```
isRed()
```

Methods assigning boolean variable can be named with verb phrases beginning with "set"/"be"

```
beOff(), setStateOffline()
```

Method names

Methods returning `void` should be named with imperative verbs describing what they must do

```
openDBLink()
```

Methods converting a value into another should be named with verb phrases starting with "as"/"to" and

denoting the converted type

```
asDecimal(), toString()
```

Other methods should describe what they return

```
previousSignal()
```

Accessor methods may report the variable’s name prefixed with “get” or “set”

Variables

Variables should be named for the objects they represent

Usually named with a **singular** noun

If it represent a **collection** of objects, its name should be **plural**

The name should not include typing information

lowerCamelCase

One lowercase letter variable names are OK only for temporary variables, indexes, etc.

```
//OK
int numberOfSheeps;
String title;
String[] books;
int i, j;

//NOT OK
long stringOfGreekLetters;
String myIdeas;
long longNumber;
int VeryLong;
```

Parameter and constant names

Name parameters in such a way that the method call is readable

```
public static double power(double base, double exponent) {
      //
}
```

Use named constants and not literal values, wherever a specific value is needed

In order to differentiate the names of constants from the other names, constants are often completely capitalized and compound names are separated by an underscore

```java
// OK
public static final double PI = 3.141593;

public static void foo(double r, double m, double h) {
    double length = PI * r*r;
    // NOT OK
    double energy = m * h * 9.81;
}
```

Code readability conventions

These conventions are often very close to design guidelines, since code readability is obtained with cohesive classes and methods

- Classes and methods focused on performing a **single task**

Methods must be short

- In Java standard less than 10 statements

Every method performs just one task, and

Every full method must fit on one screen

Formatting conventions

Very important to ease code readability and to make quicker code inspections

The specific code convention adopted is less important than sticking to the same convention throughout the code

NetBeans: Alt + Shift + F

Eclipse: Ctrl/Cmd + Shift + F

Formatting conventions

One statement per line

Use **indents** to highlight structured programming constructs

- Do not indent too much…do not waste horizontal space!
- Do not indent too little…make the structure evident!

Putting brackets

- **Open brace “{” appears at the end** of the same line as the class, interface, or method declaration
- **Closing brace “}” starts a line by itself** aligned with the opening statement, except null block “{}”
- **No space between a method name and the parenthesis “(”** starting its parameter list
- **Methods are separated by a blank line**
- When a nested statements occur within blocks
    - Use the 4 spaces rule, in general

```
int foo(int a, int b) {
    return 4;
}


double bar(double a, int b) {
    return 5.0;
}
```

Lines and wrapping

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

```
public static double veryLongMethodSignature(double can, double bring,
        double some, double problems, double fitting) {
    String x = "very long string followed by a chain".replace("R", "TTT").
            concat("QQQ").
            concat("RRR");
    double result = (123345435.65 < 5353454.0) ? 123.4
                                                 : 123.5;
    result = (123345435.65 < 5353454.0)
            ? 123.4
            : 123.5;
    return result;
}
```

Variables declaration

- There should be usually only one declaration per line to promote comments of the variables
  - Variables should be initialized where they are declared…
  - Unless the value of the variable depends on some computations to be performed later
- Declarations should be placed at the beginning of the outermost block where a variable is used
- Avoid declarations in inner blocks of variables with the same name as variable in outer blocks

Example of good style

http://www.docjar.net/html/api/java/util/Collections.java.html

Extra task

Given a system of linear equations, solve it using Cramer's rule reusing created code.