# Seminar 1
# (3-9 October 2017)

A. Please read the course rules at:
**http://www.cs.ubbcluj.ro/~craciunf/MetodeAvansateDeProgramare/CourseRules.pdf**

B. Discussion of the LAB1 implementation using the following problem: *In a box there are apples, cakes and books. Please display all entities which have their weight greater than 200 grams.*

1. We discuss the implementation of the Model, Repository, Controller and View.
2. We discuss the solution based on the class hierarchy+method overriding and the solution based on interfaces.
3. We discuss how to use packages to organize the code.
4. We discuss how to treat the errors using exceptions.

# Seminar 2
# week 2 (10-16 October 2017)

1. Discussion of the Lab-Assignment 2 implementation.
2. We discuss how the toy language interpreter works.
3. We discuss the implementation of the Model, Repository, Controller and View.
4. The discussion about generic collections is left for the Seminar 3.

# Seminar 3
# week 3 (17 -23 October 2017)

A. We continue the Discussion of the Lab-Assignment 2 implementation:

1. Discussion how the programs ASTs are directly encoded in the main method. Please write some examples (you can use the Lab-Assignment 2 examples for discussion).
2. Discussion of PrgState implementation, namely the implementation of one of those 3 ADTs. For example we can discuss ADT Stack: our interface MyIStack<T>, our class MyStack<T> implemented based on java.util.Stack<T> .
3. Discussion of other questions/problems which have arisen.

B. Discussion about generics: wildcards, bounded wildcards, raw types. Please discuss some of the examples from generics-examples.pdf (ideally all of them, if you have enough time).

# Seminar 4
# week 4 (24 – 30 October 2017)

1. **Discussion of the Lab-Assignment3 from Laboratory 4.** Please discuss how the implementation must be done. Regarding the View part you may want to discuss how it would be possible to call many times the execution of the same example (how we need to modify PrgState, Repository and Controller).

2. **Discussion of the following IO classess usage:** FileReader, FileWriter, BufferedReader, BufferedWriter, StreamTokenizer, Scanner and PrintStream. Some code templates of using these classes are given below:
   - **FileReader class example:**

```
try(FileReader fileReader = new FileReader("c:\\data\\text.txt")){
          int data = fileReader.read();
     while(data != -1) { // read a char
       System.out.print((char) data));
       data = fileReader.read();
     }
}
```

   - **FileWriter class example:**

```
try(FileWriter fileWriter = new FileWriter("data\\filewriter.txt",true)){
         //true –appends, false or nothing-overwrites
          fileWriter.write("data 1");
          fileWriter.write("data 2");
          fileWriter.write("data 3");
}
```

   - **BufferedReader class example:**

```
Reader reader = new FileReader("data.bin");
try(BufferedReader bufferedReader =new BufferedReader(reader)){
   String line = bufferedReader.readLine();
   while(line != null) {
      //do something with line

      line = bufferedReader.readLine();
   }
}
 or

br=new BufferedReader(new FileReader(numefis));
      String linie;
      while((linie=br.readLine())!=null){
         String[] elems=linie.split("[|]");
         if (elems.length<2){
            System.err.println("Linie invalida "+linie);
            continue;}
        //do something with the line
      }
```

- **BuferredWriter class example:**

```
FileWriter output = new FileWriter("data.bin");
try(BufferedWriter bufferedWriter = new BufferedWriter(output)){
   for(i=0;i<100;i++){
         bufferedWriter.write("Hello World");
         bufferedWriter.newLine();
         if(i%5==0)
            bufferedWriter.flush();
  }
}
```

- **StreamTokenizer class example:**

```
Reader reader = new FileReader("data.bin");
try(StreamTokenizer streamTokenizer = new StreamTokenizer(reader)){

   while(streamTokenizer.nextToken() != StreamTokenizer.TT_EOF){

      if(streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
         System.out.println(streamTokenizer.sval);
      } else if(streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
         System.out.println(streamTokenizer.nval);
      } else if(streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
         System.out.println();
      }
   }
}
```

- **PrintWriter class example:**

```
         FileWriter  writer      = new FileWriter("report.txt");
         PrintWriter printWriter = new PrintWriter(writer);
         printWriter.print(true);
         printWriter.print((int) 123);
         printWriter.print((float) 123.456);
         intVar i=200;
         printWriter.printf("Text + data: %d", intVar);
         printWriter.close();
```

- **Scanner class examples:**

```
Scanner sc = new Scanner(new File("myNumbers"));
     while (sc.hasNextLong()) {
        long aLong = sc.nextLong();
     }
```

# Seminar 5
# week 5 (31 October – 6 November 2017)

1. Questions from **Lab-Assignment3 from Laboratory 4.**
2. Discussion of **Lab-Assignment4 from Laboratory 5.**
3.Discuss a classical Java functional programming example of treating a text file as a stream of strings:

**3.1. First Approach**

**3.1.1. Basic idea**

```
Stream<String> lines = Files.lines(somePath);
//reading a file as a stream of strings

public static void main(String[] args) throws Exception {
        Files.lines(Paths.get("input-file"))
                .map(someFunction)
                .filter(someTest)
                .someOtherStreamOperation(...); }
```

**3.1.2. Printing out all palindromes contained in a text file**

```
public static void main(String[] args) throws Exception {
        String inputFile = "in.txt";
        Files.lines(Paths.get(inputFile))
                .filter(StringUtils::isPalindrome)
                 .forEach(System.out::println);
}

public class StringUtils {
public static String reverseString(String s) { return(new StringBuilder(s).reverse().toString()); }
public static boolean isPalindrome(String s) { return(s.equalsIgnoreCase(reverseString(s))); }
}
```

**3.2. Second Approach**

**3.2.1. Basic Idea**

```
public static void useStream(Stream<String> lines, ...) {
        lines.filter(...).map(...)...;
 }

public static void useFile(String filename, ...) {
        try(Stream<String> lines = Files.lines(Paths.get(filename))) {
                SomeClass.useStream(lines, ...);
        } catch(IOException ioe) { System.err.println("Error reading file: " + ioe); }
}
```

**3.2.2. Printing out all palindromes contained in a text file**

```
public class FileUtils {
    public static void printAllPalindromes(Stream<String> words){
        words.filter(StringUtils::isPalindrome)
                .forEach(System.out::println);
    }

   public static void printAllPalindromes(String filename) {
```

```
        try(Stream<String> words = Files.lines(Paths.get(filename))) {
                printAllPalindromes(words);
        } catch(IOException ioe) { System.err.println("Error reading file: " + ioe); }
    }
}

public static void testAllPalindromes(String filename) {
        List<String> testWords = Arrays.asList("bog", "bob", "dam", "dad");
        System.out.printf("All palindromes in list %s:%n", testWords);
        FileUtils.printAllPalindromes(testWords.stream());
        System.out.printf("All palindromes in file %s:%n", filename);
        FileUtils.printAllPalindromes(filename);
}
```

### 3.3. Third Approach

#### 3.3.1. Basic Idea:

```
public static void useStream(Stream<String> lines) {
        lines.filter(...).map(...)...;
}

public static void useFile(String filename) {
        StreamProcessor.processFile(filename, SomeClass::useStream); }

@FunctionalInterface
public interface StreamProcessor {
        void processStream(Stream<String> strings);

        public static void processFile(String filename, StreamProcessor processor) {
                try(Stream<String> lines = Files.lines(Paths.get(filename))){
                        processor.processStream(lines);
                }
                catch(IOException ioe) { System.err.println("Error reading file: " + ioe); }
        }
}
```

#### 3.3.2. Printing out all palindromes contained in a text file

```
public static void printAllPalindromes(Stream<String> words){
        words.filter(StringUtils::isPalindrome)
        .forEach(System.out::println); }

public static void printAllPalindromes(String filename) {
        StreamProcessor.processFile(filename, FileUtils::printAllPalindromes); }

public static void testAllPalindromes(String filename) {
        List<String> testWords = Arrays.asList("bog", "bob", "dam", "dad");
        System.out.printf("All palindromes in list %s:%n", testWords);
        FileUtils.printAllPalindromes(testWords.stream());
        System.out.printf("All palindromes in file %s:%n", filename);
        FileUtils.printAllPalindromes(filename); }
```

# Seminar 6 – written test
# week 6 (7 November – 13 November 2017)

1. You have to solve a simple problem in a similar manner to what you have already done at the laboratory. The test is closed-book. You must write on the paper the solution for the given problem. Each seminar group has a different problem.
2. OPTIONAL: You can implement the test problem at the laboratory.

**Seminar 7**
**week 7 (14-20 November 2017)**

1. Questions from **Lab-Assignment 4 from Laboratory 5**.
2. Discussion of the **Lab-Assignment 5 from Laboratory 7**.
3. Solve the following problems using the functional programming style (**using Java Streams):**

please start with a List of Strings similar to this:
• List<String> words = Arrays.asList("hi", "hello", ...);

**P1.** Loop down the words and print each on a separate line, with two spaces in front of each word. Don't use map.

**P2.** Repeat the previous problem, but without the two spaces in front. This is trivial if you use the same approach as in #1, so the point is to use a method reference here, as opposed to an explicit lambda in problem 1.

**P3.** We assume that we have a method StringUtils.transformedList(List<String>, Function1<String>)
where interface Function1<T> { T app(T);}
and we produced transformed lists like this:
• List<String> excitingWords = StringUtils.transformedList(words, s -> s + "!");
• List<String> eyeWords = StringUtils.transformedList(words, s -> s.replace("i", "eye"));
• List<String> upperCaseWords = StringUtils.transformedList(words, String::toUpperCase);
Produce the same lists as above, but this time use streams and the builtin "map" method.

**P4.** We assume that we have the method StringUtils.allMatches(List<String>, Predicate1<String>)
where interface Predicate1<T> { boolean check(T);}
and we produced filtered lists like this:
• List<String> shortWords = StringUtils.allMatches(words, s -> s.length() < 4);
• List<String> wordsWithB = StringUtils.allMatches(words, s -> s.contains("b"));
• List<String> evenLengthWords = StringUtils.allMatches(words, s -> (s.length() % 2) == 0);
Produce the same lists as above, but this time use "filter".

**P5.** Turn the strings into uppercase, keep only the ones that are shorter than 4 characters, of what is remaining, keep only the ones that contain "E", and print the first result. Repeat the process, except checking for a "Q" instead of an "E". When checking for the "Q", try to avoid repeating all the code from when you checked for an "E".

**P6.** Produce a single String that is the result of concatenating the uppercase versions of all of the Strings. Use a single reduce operation, without using map.

**P7.** Produce the same String as above, but this time via a map operation that turns the words into uppercase, followed by a reduce operation that concatenates them.

**P8.** Produce a String that is all the words concatenated together, but with commas in between. E.g., the result should be "hi,hello,...". Note that there is no comma at the beginning, before "hi", and also no comma at the end, after the last word. Major hint: there are two versions of reduce discussed in the notes.

**P9.** Find the total number of characters (i.e., sum of the lengths) of the strings in the List.

**P10.** Find the number of words that contain an “h”.

## 4. Discuss some of Java 9 additions to Stream:

**TakeWhile**

```
Stream.of("a", "b", "c", "", "e").takeWhile(s -> !String.isEmpty(s));forEach(System.out::print);
//Result: abd
```

**DropWhile**

```
Stream.of("a", "b", "c", "de", "f").dropWhile(s -> s.length <= 1);.forEach(System.out::print);
//Result: def
```

**Iterate from Java 8**

```
Stream.iterate(1, i -> 2 * i).forEach(System.out::println);
// output: 1 2 4 8 ...
```

**Iterate from Java 9**

```
Stream.iterate(1, i -> i <= 10, i -> 2 * i).forEach(System.out::println);
// output: 1 2 4 8
```

# Seminar 8
# week 8 (21-27 November 2017)

**NOTE: Starting from week 10 (4-8 December 2017) all groups will have the same topics for seminars and laboratories. The week will start on Monday and end on Friday. Only the groups from Tuesday and Wednesday will have the Seminar 9 and Laboratory 9 due to National Day Holiday.**

**1**. Questions from **Lab-Assignment 5 from Laboratory 7**.
**2**. Discussion of the **Lab-Assignment 6 from Laboratory 8**.
**3.** Discussion of the asynchronous execution of tasks by using ExecutorService.
Please use the following examples taken from the tutorial :
http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

Example 1:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

Example 2:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

Example 3:

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);
```

```
System.out.println("future done? " + future.isDone());
Integer result = future.get();
System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

Example 4:

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

# Seminar 9
# week 9 (28-29 November 2017)

**1**. Questions from **Lab-Assignment 5 from Laboratory 7**.
**2**. Questions from **Lab-Assignment 6 from Laboratory 8**.
**3.** Discuss how to solve the optional requirements of **Lab-Assignment 6.**
**4.** Discuss the possible solutions to the following problem: In the current **Lab-Assignment 6**, when a thread one step execution terminates with an error the entire program execution is halted. Please find a solution such that the entire program execution continues and only the thread whose onestep execution terminates with an error is halted and deleted.

**Seminar 10**
**week 10 (14-20 November 2017)**

1. Questions from **Lab-Assignment 6 from Laboratory 8**.
2. Solve some simple problems in C#. Please discuss some possible implementations for each of the following topics:

**2.1.** **An example of using properties which are declared in an interface:** Declare and implement an interface Employee with two properties: name and unique identifier.

**2.2.** **An example of using inherited properties:** Define two classes Cube and Square which implement an abstract class, Shape, and override its abstract Area property. Please use of the override modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the corresponding side for the square and cube.

**2.3.** **An example of using indexer that is declared in an interface.**

**2.4.** **An example of using delegates:** Implement a class BookDB that encapsulates a bookstore database that maintains a database of books. It exposes a method, ProcessPaperbackBooks, which finds all paperback books in the database and calls a delegate for each one. The delegate type that is used is named ProcessBookDelegate. The Test class uses this class to print the titles and average price of the paperback books.

**2.5.** **An example of using events declared in the base class:** When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

# Advanced Programming Methods

## Seminar 12

# Overview

1. instanceof operator
2. Java Serialization
3. Discuss how we can serialize our ToyLanguage interpreter. Please discuss the implementation of different interesting scenarios (one ProgramState is serialized, entire Repository is serialized, etc.)

Note: Notes are based on some online (including Oracle) tutorials.

# Instanceof operator

- compares an object to a specified type.

- to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

- null is not an instance of anything.

# Instanceof operator

```
class Parent {}

class Child extends Parent implements MyInterface {}

interface MyInterface {}


Parent obj1 = new Parent();

Parent obj2 = new Child();


obj1 instanceof Parent: true

obj1 instanceof Child: false

obj1 instanceof MyInterface: false

obj2 instanceof Parent: true

obj2 instanceof Child: true

obj2 instanceof MyInterface: true
```

# Real use of Instanceof operator

```
interface Printable{}

class A implements Printable{

public void a(){System.out.println("a method");}

}

class B implements Printable{

public void b(){System.out.println("b method");}

}

class Call{

void invoke(Printable p){

if(p instanceof A){

A a=(A)p;//Downcasting

a.a();

}

if(p instanceof B){

B b=(B)p;//Downcasting

b.b();

} } }
```

# Java Serialization

# Serializable in Java

- If you want a class object to be serializable, all you need to do it implement the java.io.Serializable interface.

- Serializable in java is a marker interface and has no fields or methods to implement.

- Serialization in java is implemented by ObjectInputStream and ObjectOutputStream, so all we need is a wrapper over them to either save it to file or send it over the network.

# An Example

```
package seminar12.serialization;

import java.io.Serializable;

public class Employee implements Serializable {

private String name;

private int id;

transient private int salary;// to be not serialized to stream

@Override

public String toString(){ return "Employee{name="+name+",id="+id+",salary="+salary+"}";}

//getter and setter methods

public String getName() {return name;}

public void setName(String name) {this.name = name;}

public int getId() {return id; }

public void setId(int id) { this.id = id;}

public int getSalary() { return salary; }

public void setSalary(int salary) {this.salary = salary;}

}
```

# General utility methods for serialization

```
package seminar12.serialization;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;


public class SerializationUtil {

// deserialize to Object from given file

public static Object deserialize(String fileName) throws IOException, ClassNotFoundException {

FileInputStream fis = new FileInputStream(fileName);

ObjectInputStream ois = new ObjectInputStream(fis);

Object obj = ois.readObject();

ois.close();

return obj;

}
```

# General utility methods for serialization

```
// serialize the given object and save it to file
public static void serialize(Object obj, String fileName) throws IOException {
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(obj);


fos.close();
}


}
```

# An example

```
/package com.journaldev.serialization;

import java.io.IOException;

public class SerializationTest {

public static void main(String[] args) {

String fileName="employee.ser";

Employee emp = new Employee();

emp.setId(100);

emp.setName("ABC");

emp.setSalary(5000);

//serialize to file

try {

SerializationUtil.serialize(emp, fileName);

} catch (IOException e) {

e.printStackTrace();

return;

}
```

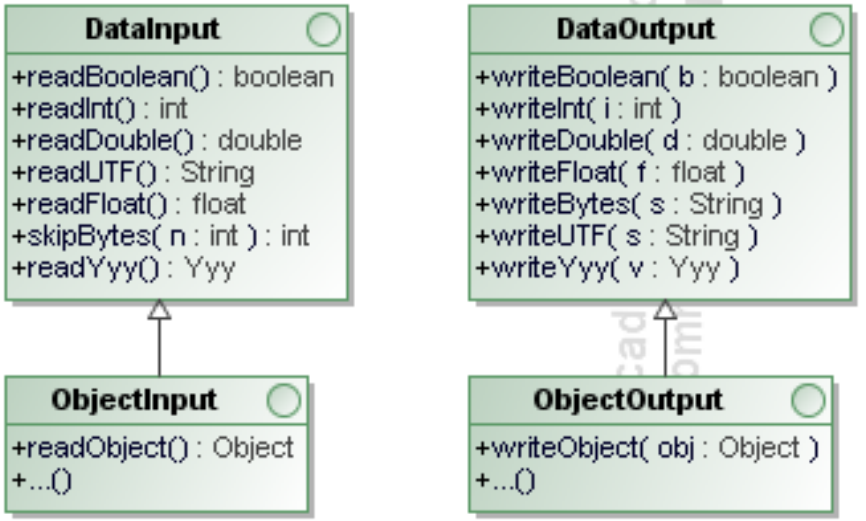
# An example

```
Employee empNew = null;

try {

empNew = (Employee) SerializationUtil.deserialize(fileName);

} catch (ClassNotFoundException | IOException e) {

e.printStackTrace();

}

System.out.println("emp Object::"+emp);

System.out.println("empNew Object::"+empNew);

}}

//OUTPUT

emp Object::Employee{name=ABC,id=100,salary=5000}

empNew Object::Employee{name=ABC,id=100,salary=0}
```

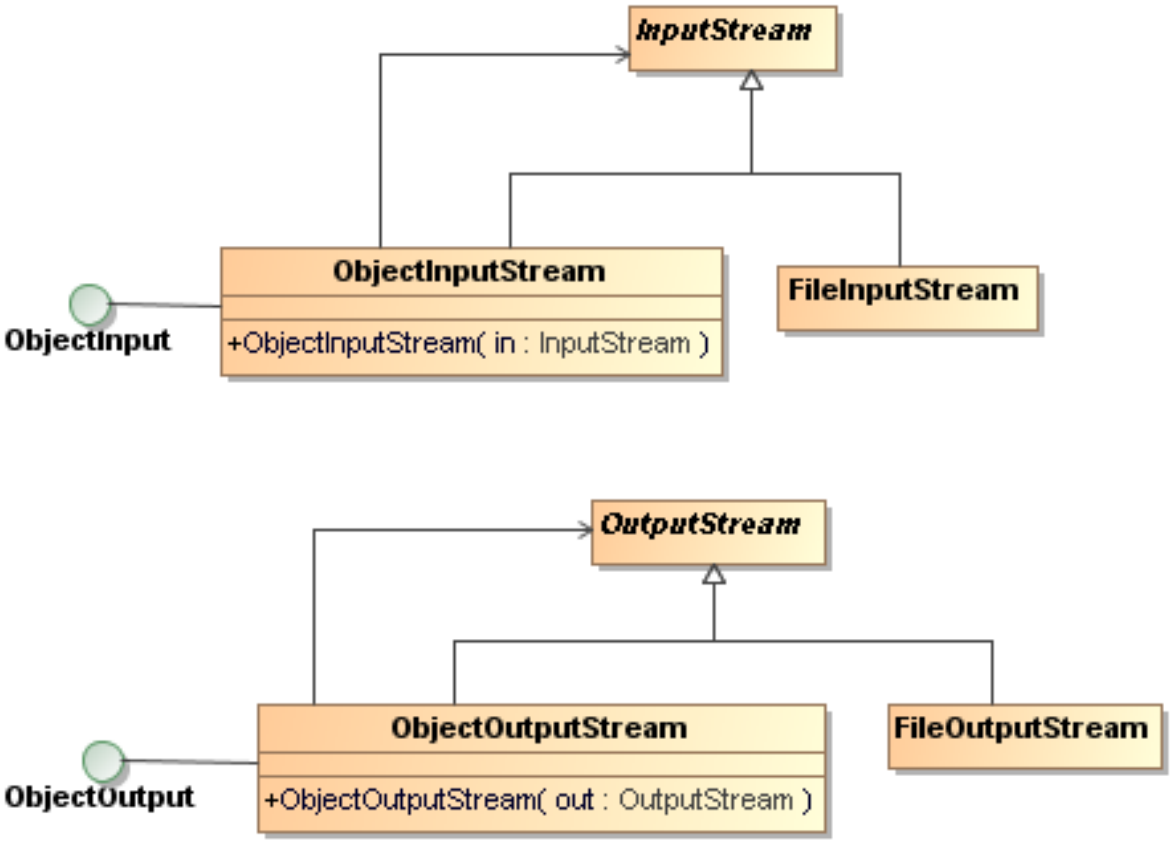
- salary is a transient variable and it’s value was not saved to file
- Similarly static variable values are also not serialized since they belongs to class and not object.

# Java Objects Serialization

- The process of  writing/reading objects from/to a file/external support.
- An object is persistent (serializable ) if it can be written into a file/external support and can be read from a file/external support



# Objects Serialization



# Class Refactoring with Serialization and serialVersionUID

Serialization in java permits some changes in the java class if they can be ignored.Some of the changes in class that will not affect the deserialization process are:

- Adding new variables to the class
- Changing the variables from transient to non-transient, for serialization it's like having a new field.
- Changing the variable from static to non-static, for serialization it's like having a new field.

For all these changes to work, the java class should have **serialVersionUID** defined for the class.

# Class Refactoring with Serialization and serialVersionUID

- If we change the Employee class as follows:

```
public class Employee implements Serializable {
…
private String password;
…
public String getPassword() { return password;}
public void setPassword(String password) { this.password = password;}
}
```

# Class Refactoring with Serialization and serialVersionUID

- And run the following:

```
public class DeserializationTest {

public static void main(String[] args) {

String fileName="employee.ser";

Employee empNew = null;

try {

empNew = (Employee) SerializationUtil.deserialize(fileName);

} catch (ClassNotFoundException | IOException e) { e.printStackTrace();}

System.out.println("empNew Object::"+empNew);

}}
```

- We got an error!! The reason is that serialVersionUID of the previous class and new class are different.
- if the class doesn't define serialVersionUID, it's getting calculated automatically and assigned to the class. Java uses class variables, methods, class name, package etc to generate this unique long number.

# Class Refactoring with Serialization and serialVersionUID

- In order to avoid the error we have to add the following field to the original Employee class

```
private static final long serialVersionUID = -6470090944414208496L;
```

- Now we will serialize it and then will add the new filed password to Employee class and will deserialize it again we will not get any error.
  - the object stream is deserialized successfully because the change in Employee class is compatible with serialization process.

# Java Externalizable Interface

- the java serialization process is done automatically.
- Sometimes we want to obscure the object data to maintain it's integrity.
- We can do this by implementing java.io.Externalizable interface and provide implementation of writeExternal() and readExternal() methods to be used in serialization process.

# Java Externalizable Interface

```
 package seminar12.externalization;

import java.io.Externalizable;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectOutput;


public class Person implements Externalizable{

private int id;

private String name;

private String gender;

@Override

public void writeExternal(ObjectOutput out) throws IOException {

out.writeInt(id);

out.writeObject(name+"xyz");

out.writeObject("abc"+gender);

}

@Override

public String toString(){ return "Person{id="+id+",name="+name+",gender="+gender+"}";}
```

# Java Externalizable Interface

```
@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
id=in.readInt();
//read in the same order as written
name=(String) in.readObject();
if(!name.endsWith("xyz")) throw new IOException("corrupted data");
name=name.substring(0, name.length()-3);
gender=(String) in.readObject();
if(!gender.startsWith("abc")) throw new IOException("corrupted data");
gender=gender.substring(3);
}
public int getId() { return id;}
public void setId(int id) {this.id = id;}
public String getName() {return name;}
public void setName(String name) {this.name = name;}
public String getGender() {return gender;}
public void setGender(String gender) {this.gender = gender;}
}
```

# Java Externalizable Interface

```
package  seminar12.externalization;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;


public class ExternalizationTest {

public static void main(String[] args) {

String fileName = "person.ser";

Person person = new Person();

person.setId(1);

person.setName("ABC");

person.setGender("Male");
```

# Java Externalizable Interface

```
try {
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(person);
    oos.close();
} catch (IOException e) {e.printStackTrace();}


FileInputStream fis;
try {
fis = new FileInputStream(fileName);
ObjectInputStream ois = new ObjectInputStream(fis);
    Person p = (Person)ois.readObject();
    ois.close();
    System.out.println("Person Object Read="+p);
} catch (IOException | ClassNotFoundException e) { e.printStackTrace();}
 }}
```

# Java Serialization Methods

- serialization in java is automatic and all we need is implementing Serializable interface
- there are four methods that we can provide in the class to change the serialization behavior:
    - readObject(ObjectInputStream ois): If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from stream.
    - writeObject(ObjectOutputStream oos): If this method is present in the class, ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usage is to obscure the object variables to maintain data integrity.
    - Object writeReplace(): If this method is present, then after serialization process this method is called and the object returned is serialized to the stream.
    - Object readResolve(): If this method is present, then after deserialization process, this method is called to return the final object to the caller program.

# Serialization with Inheritance

- Sometimes we need to extend a class that doesn't implement Serializable interface.
- If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.
- This is one place, where readObject() and writeObject() methods really help. By providing their implementation, we can save the super class state to the stream and then retrieve it later on.
- See the following example:

# Serialization with Inheritance

- Superclass does not implement Serializable

```
package seminar12.serialization.inheritance;


public class SuperClass {

private int id;

private String value;

public int getId() {

return id;}

public void setId(int id) {

this.id = id;}

public String getValue() {

return value;}

public void setValue(String value) {

this.value = value;}

}
```

# Serialization with Inheritance

```
package seminar12.serialization.inheritance;


import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;


public class SubClass extends SuperClass implements Serializable, ObjectInputValidation{
private static final long serialVersionUID = -13223221399263903291;
private String name;


public String getName() {return name;}
public void setName(String name) {this.name = name;}
@Override
public String toString(){return "SubClass{id="+getId()+",value="+getValue()+",name="+getName()+"}";}
```

# Serialization with Inheritance

```
//adding helper method for serialization to save/initialize super class state
private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException{
ois.defaultReadObject();
//notice the order of read and write should be same
setId(ois.readInt());
setValue((String) ois.readObject());        }

private void writeObject(ObjectOutputStream oos) throws IOException{
oos.defaultWriteObject();
oos.writeInt(getId());
oos.writeObject(getValue());}

@Override
public void validateObject() throws InvalidObjectException {
//validate the object here
if(name == null || "".equals(name)) throw new InvalidObjectException("name can't be null or empty");
if(getId() <=0) throw new InvalidObjectException("ID can't be negative or zero");
}     }
```

# Serialization with Inheritance

```
package seminar12.serialization.inheritance;

import java.io.IOException;

import seminar12.serialization.SerializationUtil;

public class InheritanceSerializationTest {

public static void main(String[] args) {

String fileName = "subclass.ser";

SubClass subClass = new SubClass();

subClass.setId(10);

subClass.setValue("Data");

subClass.setName("ABC");

try {

SerializationUtil.serialize(subClass, fileName);

} catch (IOException e) {e.printStackTrace();return;}

try {

SubClass subNew = (SubClass) SerializationUtil.deserialize(fileName);

System.out.println("SubClass read = "+subNew);

} catch (ClassNotFoundException | IOException e) {e.printStackTrace();}}

}
```

# Serialization Proxy Pattern

- Java Serialization pitfalls:
    - The class structure can't be changed a lot without breaking the java serialization process. So even though we don't need some variables later on, we need to keep them just for backward compatibility.
    - Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, user role is serialized and an attacker change the stream value to make it admin and run malicious code.

# Serialization Proxy Pattern

- Serialization Proxy pattern is a way to achieve greater security with Serialization.

  - an inner private static class is used as a proxy class for serialization purpose. This class is designed in the way to maintain the state of the main class.

  - This pattern is implemented by properly implementing readResolve() and writeReplace() methods.

  - See the following example

# Serialization Proxy Pattern

```
package seminar12.serialization.proxy;

import java.io.InvalidObjectException;

import java.io.ObjectInputStream;

import java.io.Serializable;


public class Data implements Serializable{

private static final long serialVersionUID = 2087368867376448459L;

private String data;


public Data(String d){ this.data=d;}


public String getData() {return data;}


public void setData(String data) {this.data = data;}


@Override

public String toString(){return "Data{data="+data+"}";

}
```

# Serialization Proxy Pattern

```
//serialization proxy class

private static class DataProxy implements Serializable{

private static final long serialVersionUID = 8333905273185436744L;

private String dataProxy;

private static final String PREFIX = "ABC";

private static final String SUFFIX = "DEFG";


public DataProxy(Data d){

//obscuring data for security

this.dataProxy = PREFIX + d.data + SUFFIX;}


private Object readResolve() throws InvalidObjectException {

if(dataProxy.startsWith(PREFIX) && dataProxy.endsWith(SUFFIX)){

return new Data(dataProxy.substring(3, dataProxy.length() -4));

}else throw new InvalidObjectException("data corrupted");

}

}
```

# Serialization Proxy Pattern

```
//replacing serialized object to DataProxy object
private Object writeReplace(){
return new DataProxy(this);
}

private void readObject(ObjectInputStream ois) throws InvalidObjectException{
throw new InvalidObjectException("Proxy is not used, something fishy");
}
}
```

# Serialization Proxy Pattern

- Data and DataProxy class should implement Serializable interface.
- DataProxy should be able to maintain the state of Data object.
- DataProxy is inner private static class, so that other classes can't access it.
- DataProxy should have a single constructor that takes Data as argument.

- Data class should provide writeReplace() method returning DataProxy instance. So when Data object is serialized, the returned stream is of DataProxy class. However DataProxy class is not visible outside, so it can't be used directly.
- DataProxy class should implement readResolve() method returning Data object. So when Data class is deserialized, internally DataProxy is deserialized and when it's readResolve() method is called, we get Data object.
- Finally implement readObject() method in Data class and throw InvalidObjectException to avoid hackers attack trying to fabricate Data object stream and parse it.

# Serialization Proxy Pattern

```
package seminar12.serialization.proxy;

import java.io.IOException;

import seminar12.serialization.SerializationUtil;


public class SerializationProxyTest {

public static void main(String[] args) {

String fileName = "data.ser";

Data data = new Data("ABC");

try {

SerializationUtil.serialize(data, fileName);

} catch (IOException e) {e.printStackTrace();}

try {

Data newData = (Data) SerializationUtil.deserialize(fileName);

System.out.println(newData);

} catch (ClassNotFoundException | IOException e) { e.printStackTrace();}

}

}
```

# Serializable objects

- All the reachable objects (the objects that can be reach using the references) are saved into the file only once.

```
class CircularList implements Serializable{
  private class Node implements Serializable{
      Node urm;
      //...
  }
   private Node head; //last node of the list refers to the head of the list
   //...
}
```

- The objects which are referred by a serializable object must be also serializable.

# Serializable data structures

```
public class Stack implements Serializable{
   private class Node implements Serializable{
   //...
   }
    private Node top;
   //...
 }
//...
Stack s=new Stack();
 s.push("ana");
 s.push(new Produs("Paine", 2.3));
                  //class Produs must be serializable
//...
 ObjectOuputStream out=...
  out.writeObject(s);
```

# Advanced Programming Methods

## Seminar 13-14

# Overview

1. Exam rules
2. Theoretical Exam -sample
3. Practical Exam - sample

# Grading

**Laboratory activity:-- 30%**

- 8 Lab-Assignments having the same importance. The laboratory grade is the arithmetic average of those individual 8 grades.

**Seminar activity (including a test) – 5%**

- seminar test is 5%
- your seminar contribution will be taken into account for the rounding of the final mark

**Final exam: -- 65%**

- Final Written Exam (about 1 hour, closed books): **--15%**
- Final Practical Exam (about 3 hours, open books): **--50%**

# Rules to enter into the Final Exam

- in order to get into the final exam you have to attend minimum 90% of the labs and minimum 70% of the seminars. That means you must attend minimum 10 seminars and minimum 12 laboratories. Please read the following document:

http://www.cs.ubbcluj.ro/wp-content/uploads/Hotarare-CDI-15.03.2017.pdf

# Final Exam Rules

- the final practical exam:

  -- 3 hours, open book (you can access your projects, lecture notes, seminar notes, java manuals and tutorials)

  -- you have to work on your Java implementations of your interpreter to add more functionalities

- the final written exam:

  -- closed book, 1 hour, some general questions from lecture notes

- **in order to pass the final exam you must have:**

  **-- at least 5 at the final written exam and**

  **-- at least 5 at the final practical exam and**

  **-- the final grade must be at least 5**

- you can pass either both the final written exam and the final practical exam or nothing

# Rules for the second exam (“restanta”)

- the content and the structure are the same as those for the normal final exam (you have to work on your own java implementations of the toy language)

- **in order to pass the final second exam you must have:**

  **--- at least 5 at the final written exam and**

  **--- at least 5 at the final practical exam and**

  **--- the final grade is 5**

- you can pass either both the second final written exam and the second final practical exam or nothing

# Rules for the Students from previous years (“Restantieri”)

- the students must attend the labs and the seminars, must do the lab assignments, and must pass the final exam
- the rules (for getting into the final exam) of the laboratories and seminars attendance are not compulsory for the students “restantieri”. They can get into the final exam without any lab and seminar attendance
- the same rules are applied for the Erasmus students

# Theoretical Exam

1. It is a closed book written exam.
2. Its duration is about 45 minutes.
3. Exam requirements: Java lectures notes, Java seminars and Java laboratories from week 1 to week 12 (see lecture 13 discussion)
4. After you will complete the theoretical exam it will be a 15 minutes break before the practical exam will start. You can use that 15min break as part of the practical exam.

# Theoretical Exam - sample

**1.(3p)**. Discuss the overloading mechanism in Java.

# Theoretical Exam - sample

**2.(2p).** Given the following four classes in Java:

class A {...}  class B extends A {...}  class C extends A {...}

class Amain{

**... method1(... list) { return list.get(1);}**

**boolean method2(... list, ... el) { return list.add(el);}**

}

Discuss line by line the correctness of the following short program. Compute the most specific signatures for the class Amain methods (method1 and method2) and the most specific types for the variables "elem", "list1" and "list2" such that the following program is correct. If it is not possible to find the types justify your answer.

# Theoretical Exam - sample

ArrayList<A> listA=new ArrayList<A>(); listA.add(new A());listA.add(new A());

ArrayList<B> listB = new ArrayList<B>(); listB.add(new B());listB.add(new B());

ArrayList<C> listC = new ArrayList<C>(); listC.add(new C());listC.add(new C());

Amain ob = new Amain();

ob.method1(listA); ob.method1(listB); ob.method1(listC);

**... elem = new ...();**

ob.method2(listA,elem); ob.method2(listB,elem); ob.method2(listC,elem);

**...list1;**

list1=listB;list1=listC; ob.method1(list1);

**...list2;**

list2=listA;list2=listB; ob.method2(list2,elem);

# Theoretical Exam - sample

**3.(2p).** What is a CyclicBarrier in Java.

**4.(2p).** Expalin the behaviour of  "instanceof" in Java.

# Practical Exam

- It takes 3 hours.
- It is an open book exam.
- Please come with your laptop and your final version of your Java project.

# Practical Exam - sample

**Practical Subjects – 25 January 2017**

**Work Time: 3 hours**

**Please implement in Java the following two problems.**

**If a problem implementation does not compile or does not run you will get 0 points for that problem (that means no default points)!!!**

**If for one problem you have only a text interface to display the program execution you are penalized with 1.25 points for that problem.**

# Practical Exam - sample

1. **(0.5p by default). Problem 1:Implement lock mechanism in ToyLanguage.**

**a. (0.5p).** Inside PrgState, define a new global table (global means it is similar to Heap, FileTable and Out tables and it is shared among different threads), LockTable that maps integer to integer. LockTable must be supported by all previous statements. It must be implemented in the same manner as Heap, namely an interface and a class which implements the interface.

**b. (0.75p).** Define a new statement newLock(var) which creates a new lock into the LockTable. The statement evaluation rule is as follows:

Stack1={newLock(var)| Stmt2|...}

SymTable1

Out1

Heap1

FileTable1

LockTable1

==>

# Practical Exam - sample

Stack2={Stmt2|...}

Out2=Out1

Heap2=Heap1

FileTable2=FileTable1

LockTable2 = LockTable1 synchronizedUnion {newfreelocation ->-1}

*if* var exists in SymTable1 *then*

SymTable2 = update(SymTable1,var, newfreelocation)

*else* SymTable2 = add(SymTable1,var, newfreelocation)

Note that you must use the lock mechanisms of the host language Java over the LockTable in order to add a new lock to the table.

# Practical Exam - sample

**c. (0.75p).** Define the new statement

lock(var)

where var represents a variable from SymTable which is mapped to an index into the LockTable. Its execution on the ExeStack is the following:

- pop the statement

- foundIndex=lookup(SymTable,var). If var is not in SymTable print an error message and terminate the execution.

- *if* foundIndex is not an index in the LockTable *then* print an error message and terminate the execution

*elseif* LockTable[foundIndex]==-1 *then* LockTable[foundIndex]=Identifier of the PrgState

*else* push back the lock statement(that means other PrgState holds the lock)

Note that the lookup and the update of the LockTable must be an atomic operation, that means they cannot be interrupted by the execution of the other PrgStates. Therefore you must use the lock mechanisms of the host language Java over the LockTable in order to read and write the values of the LockTable entrances .

# Practical Exam - sample

**d. (0.75p)** Define the new statement:

unlock(var)

where var represents a variable from SymTable which is mapped to an index into the LockTable. Its execution on the ExeStack is the following:

- pop the statement

- foundIndex=lookup(SymTable,var). If var is not in SymTable print an error message and terminate the execution.

- *if* foundIndex is not an index in the LockTable *then* do nothing

*elseif* LockTable[foundIndex]== Identifier of the PrgState *then* LockTable[foundIndex]= -1

*else* do nothing

Note that the lookup and the update of the LockTable must be an atomic operation, that means they cannot be interrupted by the execution of the other PrgStates. Therefore you must use the lock mechanisms of the host language Java over the LockTable in order to read and write the values of the LockTable entrances .

# Practical Exam - sample

**e. (1p)** Extend your GUI to suport step-by-step execution of the new added features. To represent the LockTable please use a TableView with two columns  location and value.

**f. (0.75p).** Show the step-by-step execution of the following program. At each step display the content of each program state (all the structures of the program state). The step-by-step execution must be displayed on the screen and also must be saved into a readable log text file. The following program must be hard coded in your implementation.

```
new(v1,20);new(v2,30);newLock(x);

fork(

   fork(

     lock(x);wh(v1,rh(v1)-1);unlock(x)

   );

   lock(x);wh(v1,rh(v1)+1);unlock(x)

);newLock(q);
```

# Practical Exam - sample

```
fork(
   fork(lock(q);wh(v2,rh(v2)+5);unlock(q));
  m=100;lock(q);wh(v2,rh(v2)+1);unlock(q)
);
z=200;z=300;z=400;
lock(x);
print (rh(v1));
unlock(x);
lock(q);
 print(rh(v2));
unlock(q)
```

The final Out should be {20,36}

# Practical Exam - sample

**2. (0.5p by default) Problem 2:** Implement For statement in Toy Language.

**a. (2.75p).** Define the new statement:

for(v=exp1;v<exp2;v=exp3) stmt

Its execution on the ExeStack is the following:

- pop the statement

- create the following statement: v=exp1;(while(v<exp2) stmt;v=exp3)

- push the new statement on the stack

**b. (1.75p).** Show the step-by-step execution of the following program. At each step display the content of each program state (all the structures of the program state). The step-by-step execution must be displayed on the screen and also must be saved into a readable log text file. The following program must be hard coded in your implementation:

v=20;

(for(v=0;v<3;v=v+1) fork(print(v);v=v+1) );

print(v*10)

The final Out should be {0,1,2,30}