Java packages, Exceptions handling in Java

Objectives:
- defining and using packages**;**
- understanding the exceptions handling mechanisms; simple exceptions; multiple exceptions;
- throw vs. throws;
- metods that launch exceptions;
- user defined exception classes;
- assertions;

Java packages

A Java package is a collection of classes and interfaces generally grouped by functionality in order to benefit from the protection offered by this type of grouping.
A Java package is declared using the keyword *package*. The declaration of a Java package must be made before any other statement in the program (before the import section of other packages and classes from the program's header).

Syntax:
```
package package_name;
```

The source and bytecode files resulting from compiling the components of a packet must be stored in a subdirectory bearing the package name. Thus, they will be found by the JVM when importing the package into third-party applications (of course, in compliance with all the rules regarding *the classpath*).
The packages facilitate an interaction by association between the member classes, the classes in the same package being accessible to each other *(friendly access).*

Java exceptions

An exception is not a compiling error and is given by an abnormal situation that can occur during the execution of a Java application (stand-alone program, distributed application, etc.) asking the application:

- to immediately stop the execution (in case of an error and in the absence of a mechanism for managing the situation occurred)
- to execute a special action with that exception (avoiding the program blocking, if a way to manage the situation has been provided)

Most exceptions that are encountered occur in response to some error conditions such as:

- dividing an integer by zero
- stepping out the scope of a variable
- impossible access to the elements of an array
- etc.

When the interpreter encounters such a situation, an exception object is created that registers the state of the application at the time it occurred.
The exception will cause an execution jump to an area identified by an exception indicator (handler), depending on the nature of the exception. If no indicator is defined for a particular exception, the execution of the program jumps to the end of the method in which the exception occurred. This mechanism can continue, thus climbing through the entire hierarchy of methods of the application, until it reaches the top of it (e.g. *the main()* method).
The effect is stopping the current software process and even the total shutdown of the application.

Exceptions of error type

Are subclasses of the `java.lang.Error`.

These are usually not in the responsibility of programmers and often constitute system errors (memory overruns, etc.)

Specific exceptions

This category includes problems that may occur in a correct program that contains incorrect data in the given context. These exceptions are subclasses of the `java.lang.Exception` (except `java.lang.RuntimeException`).

Runtime exceptions

Here are usually programming bugs (indexes of strings outside the limits, uninitialized variables, etc.) and are represented by the class `java.lang.RuntimeException` and its subclasses. They do not necessarily have to be treated by the user, but it is recommended to be done. Such exceptions are named *unchecked* exceptions.

The exceptions generated by the Input/Output operations as well as those appeared in the Java AWT graphics (etc.) must be handled by the programmer on a mandatory basis, otherwise the compiler will issue an error. Such exceptions are named *checked* exceptions.

Class `java.lang.Throwable` is the base class for all Java exceptions.
The methods from `java.lang.Throwable` are propagated through the entire hierarchy of exception classes and are essential for the functioning of any exception handling mechanism.

Constructors:

- `Throwable()`

=> builds a new object with a *null* error message.

- `Throwable(String message)`

=> builds an object that has the error message specified as parameter

- `Throwable(String message, Throwable cause)`

=> builds an object that has the error message and cause specified as parameters

- `Throwable(Throwable cause)`

=> builds an object that has the cause specified as parameter

Member methods:

- `Throwable fillInStackTrace()`

=> fills the program's execution stack

- `Throwable getCause()`

=> returns the cause of the exception or *null* if the cause is nonexistent or unknown

- `String getLocalizedMessage()`

=> creates a localized description of the exception

- `String getMessage()`

=> returns a detailed message describing the current exception

- `StackTraceElement[] getStackTrace()`

=> provides access to the elements memorized in the program's execution stack

- `Throwable initCause(Throwable cause)`

=> initializes the cause of the current object with the value of the received parameter

- `void printStackTrace()`

=> prints (in the standard error flow) the information stored in the program's execution stack

- `void printStackTrace(PrintStream s)`

=> prints the information stored in the program execution stack in the output data stream received as a parameter

- `void printStackTrace(PrintWriter s)`

=> prints the information stored in the program execution stack in the specified *writer* output data stream

- `void setStackTrace(StackTraceElement[] stackTrace)`

=> sets the items that will be returned or printed later while calling *getStackTrace()* and *printStackTrace()*

- `String toString()`

=> returns a textual description of the current object

Exception handling mechanisms

To intercept and treat an exception, the *try/catch/finally* sequence is used.

Syntax:

```
try{
      //code that can generate exceptions
}
catch(ExceptionType1 exceptionObject){
      //exception handling block
}
catch(ExceptionType2 exceptionObject){
      //exception handling block
}
//...
catch(ExceptionTypeN exceptionObject){
      //exception handling block
}
finally{
      //code that will be executed anyway
 }
```

There are several rules governing the handling of multiple exceptions:

- exceptions of more specific types must be placed before the most general ones
- a single *catch* block (the first one that matches the occurred exception) will be executed
- several exceptions can be specified in the same *catch* block, if delimited by the OR (|) operator. *(try { … } catch(ExceptionType1 | ExceptionType2 e) {…})*
- local resources can be defined in the guard zone (*try…catch( )*). This mechanism is called *try with resources*.
  try (BufferedReader reader = …) {
  doSomeIOWith(reader);
  …
  } catch(…) {…}

Exception delegation mechanism in Java

There are situations in which the delegation of exceptions is allowed, i.e. "signaling-throwing-launching" them explicitly. The instruction `throw` is used for launching the exception, and the keyword `throws` mentioned in a method's prototype indicates that the method can launch exceptions.

Overridding methods and handling exceptions

If a derived class overrides a base class method, in addition to restrictions imposed on visibility specifiers, Java specifies that methods that override the original methods cannot signal exceptions other than those of the original method. However, it is allowed to declare a method that does not signal any exceptions, although the original method did so.

User defined exception classes

The programmer is free to define his own exception classes and then to use them as needed.

The most common mechanism is subclassing an existing exception class. Any suitable exception class, including `java.lang.Throwable` can be chosen for subclassing.

Assertions

Java Assertion = Boolean expression that is assumed to be true at the time of execution.
By default asssrtions are disabled and are most commonly used during development and testing.

Usage:

assert expression1; where *expression1* has boolean value

assert expresssion1 : expresssion2; where *expression1* has boolean value and *expression2* is an expression that has a value or a function that returns anything else but *void*; *expression2* will be executed if *expression1==false*;

OBS: Assertions are enabled at runtime:

> java -ea p1

In Eclipse, the following setting must be implemented: *VM arguments –ea*

When the assertion is evaluated as *false*, an *AssertionError* exception will be generated (unchecked exception).

A correct program should not generate *AssertionErrors*. These errors should only indicate implementation bugs. In such situations the program should stop, the error message should be seen so that the source of the problem can be located.

Assertions are used for verifying:

- if a value is in a specific domain
- if a certain code sequence is executed
- if the state of an object respects a certain constraint
- what must be true before and after calling a method

Exemple

Ex.1 defining a package

```
package pack1;   // source file: KeyboardReader.java, localized in
                 // pack1 directory

import java.util.Scanner;

public class KeyboardReader{

      Scanner scanner;

      public KeyboardReader(){
            scanner = new Scanner(System.in);
      }

      public int readInt(){
            int rez;

            System.out.print("Scrie o valoare intreaga: ");
            rez = scanner.nextInt();

            return rez;
      }

      public void close(){
            scanner.close();
      }
}
```

Ex. 1 using a package

```
import pack1.KeyboardReader;

class Test {

      public static void main(String... args){
            KeyboardReader reader = new KeyboardReader();

            int a = reader.readInt();
            System.out.println("Valoarea citita: " + a);


            reader.close();

         }
}
```

Ex. 2 Metods that launch exeptions

```
public class Exceptions_01{
      public float metodaPericuloasa(int x,int j) throws ArithmeticException{
```

```
            //float rez = (float)x/j; //Infinity
            float rez = x/j;

            return rez;
      }

      public static void main (String[] args){
            int y = 3;
            float rez;

            Exceptions_01 ob = new Exceptions_01();

            System.out.println("Inceput program");

            for(int i=-2;i<3;i++){
                  rez = 0;
                  boolean ok = true;
                  try{
                        rez = ob.metodaPericuloasa (y,i);
                  }
                  catch(ArithmeticException e){
                        ok = false;
                        System.out .println("Impartire cu zero");
                  }
                  if(ok)
                        System.out.println(y + " : " + i + " = "+rez);
            }
      }
}
```

Ex. 3 Multiple catch blocks

```
class Test {

      public void suma(int n, Scanner scanner) throws NumberFormatException,
ArrayIndexOutOfBoundsException{

            int values[] = new int[3];
            int s = 0;
            for(int i=0; i<n; i++){
                  values[i] = Integer.parseInt(scanner.nextLine());
            }

            for(int i=0; i<n; i++){
                  s += values[i];
            }

            System.out.println("Rezultatul adunarii: "+s);
      }

      public static void main(String... args){
```

```
            int n;
            Test tester = new Test();
            Scanner scanner = new Scanner(System.in);

            System.out.println("Cate valori adunam? ");
            n = Integer.parseInt(scanner.nextLine());

            try {
                  tester.suma(n, scanner);

            }
            catch (NumberFormatException e) {
                  System.out.println("Valoare introdusa in format incorect...");
            }
            catch (ArrayIndexOutOfBoundsException e) {
               System.out.println("Prea multe valori pentru un sir prea mic...");
            }
            finally {
                  System.out.println("Finally va fi rulat oricum.");
                  }
        }
}
```

Ex. 4 User defined exception classes

```
import java.lang.Throwable;

class MyException extends Throwable{
      //constructor
      public MyException(){
            //apelul constructorului clasei de baza
            super("Sirul de caractere nu are voie sa fie abc!!!");
      }
}

class X{
      void metoda(String text) throws MyException{
            if(text.equalsIgnoreCase("abc")){
                  throw new MyException();
            }
            else{
                  System.out.println("Sirul de caractere "+text+" corespunde.");
            }
      }
}

class Test{
      public static void main(String args[]){
            String text_interzis = new String("abc");
            String text_acceptat = new String("xyz");

            X ob1 = new X();
```

```
            try{
                  ob1.metoda(text_interzis);
            }
            catch(MyException ex1){
                  ex1.printStackTrace();
            }

            try{
                  ob1.metoda(text_acceptat);
            }
            catch(MyException ex2){
                  ex2.toString();
            }
      }
}
```

Ex. 5 Assestions (Eclipse setting: *VM arguments –ea, command line setting :> java -ea AssertD*)

```
import java.util.Scanner;

public class AssertD{

public static void main( String args[ ] ){
      System.out.print( "Enter a number between 0 and 10: " );
      @SuppressWarnings("resource")
      int number = new Scanner( System.in ).nextInt();
      //assert that the absolute value is >= 0 and less then 10 as internal
      //invariant
      assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
            //for false
      System.out.printf( "You entered %d\n", number );
} // end main
} // end class
```

Individual work

Complexity *

1. Write a Java program that defines an array of double values and read the appropriate data from the keyboard. Handle the exception produced when the code tries to acces an element that has a negative index or an index greater than the maximum number of elements (*ArrayIndexOutOfBoundsException*). Display a significant message when the exception occurs.
Consider a matrix with a fixed number of elements for each line and protect the code against the exception mentioned above. Consider also the case if each line has a different number of elements.

2. Write a Java application which, within the main() method, contains a squence of code that may throw various exceptions, such as *ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException*, as well as others which you consider to be useful for testing. In the catch block show the corresponding message generated by these exceptions. The finally block just prints the message, "I may or may not have caught an exception".

3. Define a package that declares an interface named *Int1* (2 integer variables and a *sum()* method that returns the sum of 2 integer values). Include in the same package a class named *Class1* (2 protected double variables, constructor, setters and getters). In another source file, add in the same package a new interface named *Int2* (2 double variables and a method named *product()* that returns the product of 2 double values).
Implement a distinct source file and import everything from the defined package. Define a class named *Class2* that is derived from *Class1* and implements both interfaces *Int1* and *Int2*.
Instantiate *Class2* and call the defined methods for determining the sum and product of some values read from the keyboard.

4. Write a Java class derived from the *Exception* class, called *SuperException*. Another class, called *SmallerException* is derived from *SuperException*. Within the classes' constructors print a message which indicates which exception was generated. In a third class create a method *a()* which throws an exception of type *SmallerException*, and a method *b()* which throws a *SuperException*.

In the *main()* method call these two methods and try to determine the type of exception which occurs, as well as if the corresponding catch block for the *SmallerException* can catch a *SuperException*.

Complexity **

5. Write an application which checks if 3 random points form an obtuse triangle. If the condition is not met, a specific exception is thrown: *AcuteTriangle, RightTriangle*. If the 3 points are on the same line or if the segments determined by the 3 points cannot make up a triangle, throw an *ImpossibleTriangle* exception, and within the corresponding catch block print a warning and throw a *RuntimeException*.

6. Write a Java application which implements an exception called *OverSaturated*. This exception is generated when the saturation of a color has a value over 90 in the HSV color space. Write a method which randomly generated colors in the RGB space and then converts them to the HSB/HSV space (https://www.cs.rit.edu/~ncs/color/t_convert.html). If after the conversion, the color's saturation is over 90, regenerate the color (In the testing phase, use an assertion to verify this internal invariant condition). After 10 consecutive tries to regenerate, throw an exception.

7. Define a Java package named *imageProcessor* which contains a class called *MyImage*. The class has all the necessary methods used for initializing and modifying the values from a m x n pixels matrix. Each pixel is an instance of another

class named *Pixel* (also included in the package) which contains 3 integer variables R, G and B with possible values between 0 and 255.
The class *MyImage* defines methods for:
- cancelling the pixels that have the RGB values below some values received as parameters
- deleting the R G or B components from all the pixels
- transforming the pixels into grayscale tones by using the formula 0.21 R + 0.71 G + 0.07 B. The new R G and B components will be equal with this formula's results.

Note: each operation is timed.

Import the defined package into a Java application that creates a *MyImage* instance. The program generates randomly the values for the pixels' components. Apply the methods stored inside the class upon the created instance. Display the results and the necessary amount of time specific to each operation.

Complexity ***

8. Write an application which checks the Romanian vehicle registration numbers. Their format is the following: [L{L}][NN{N}][LLL], where L represents a letter, N a digit, and the curly braces represent the fact that for Bucharest the number is composed of a single letter in the first group, and the digit group can be composed of 3 digits. Implement a method which checks the registration numbers and throw exceptions (instances of specialized exception classes) specific to each error which may occur upon check-up (specialized messages). For example, if the county letters group is composed of 2 letters, the digit group cannot be of size 3. The last letters group cannot contain "I" and "O" on the first and last position.

9. Define a class called *Position* which controls the position of a point in space (point_name (String), X,Y,Z coordinates (Integer), constructors, setters, getters). Declare an array of maximum 3 objects of type *Position.* The user is then asked to enter from the keyboard the number of desired objects. In the initialization sequence handle the *ArrayIndexOutOfBoundsException* generated by an incorrect number of points asked for by the user.

Test all the coordinates of the entered points, and if any of these are closer to another point hard-coded in the program, an exception of type *PointTooClose* is thrown. In the corresponding catch block the user is asked to re-enter the point's coordinates until the data is considered to be correct.

10. Consider a package of classes and interfaces called *dbInteraction* which enables the interaction with a database based on a user's authentication. The package includes the following components:
- a class which defines objects of type *Person* with the private attributes: name, surname, e-mail address, userID and password along with the getter and setter methods
- an interface for authentication with the methods *createUser(), deleteUser()* and *login().* The methods take as intput a *Person* object and return an array of *Person* variables. For the *login()* method there is a default list of users: admin, dbAdmin and superUser.
- an abstract class *VerifyPerson* which extends the *Person* class and implements the methods which check the formats of the name, surname and e-mail address with the following specs:
    * the name and surname ca only contain alphabetic characters
    * the length of the name and surname cannot be greater than 50 characters
    * the e-mail address should be formatted as: [a-zA-Z._]@[a-zA-Z.].[a-zA-Z]{2-5}
- the abstract class declares but does not implement the the methods which check the *userID* and the *password*

Create an application which interacts with this package like so:
- Write a class which implements the authentication interface and which updates the list of persons accordingly
- Write a class which extends the abstract class Verify and implements the check methods for userID and password according to the following specs:
    - The userID can only contain alfanumeric symbols plus the "." symbol
    - The password should be at least 8 characters long, with at lease one uppercase symbol and a non-alphanumeric symbol
- Write a *Test* class which tests if all the package's functionalities are correct.