Advanced Programming Methods
Lecture 5

Content

- Exceptions in Java and C#
- Java packages
- C# namespaces

Exceptions

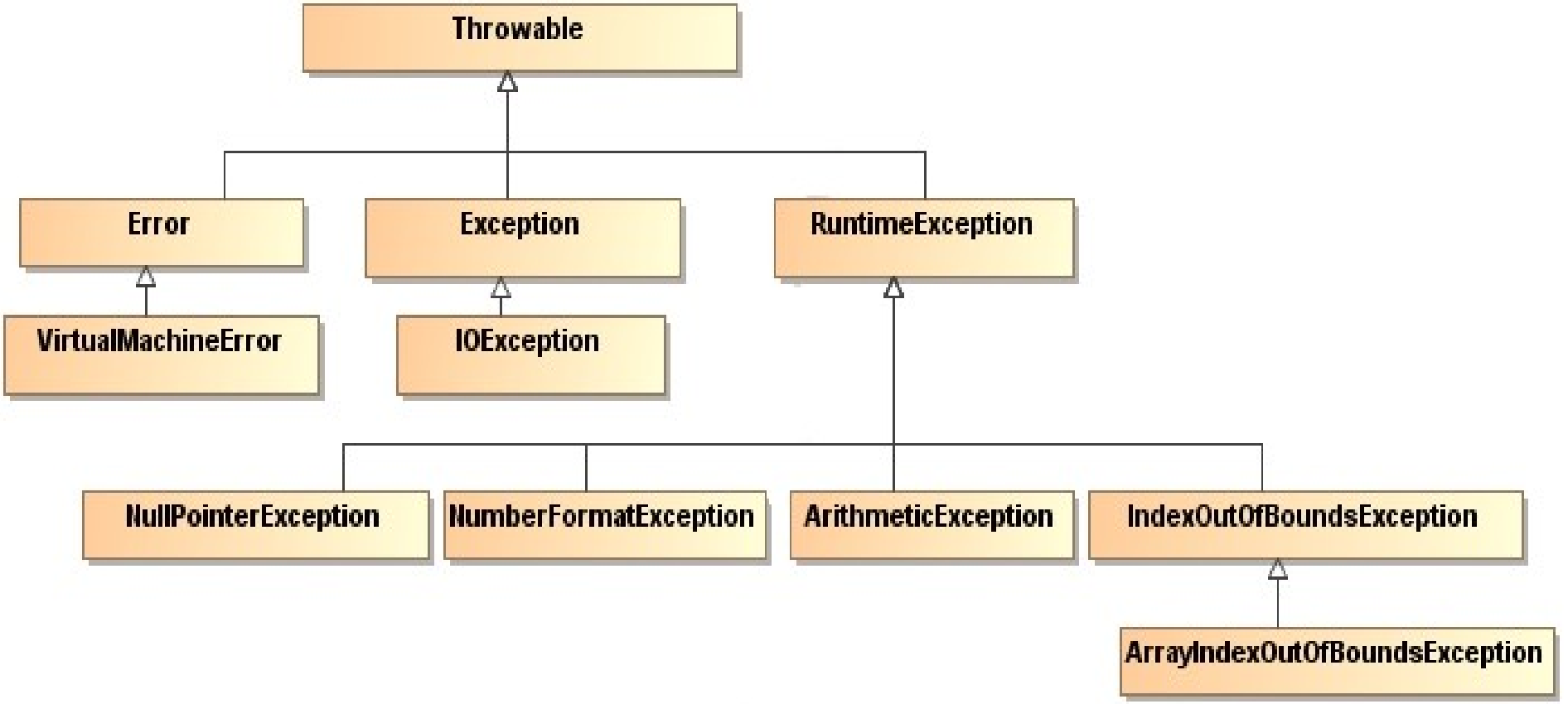
An exception is an abnormal situation that can occur during the program execution

A Java or C# exception is an object which describes an abnormal situation that may occur during the program execution

JAVA EXCEPTIONS

Java Exceptions

Three types of exceptions:Errors(external to the application), Checked Exceptions(subject to try-catch), and Runtime Exceptions(correspond to some bugs)



Example 1

Program for ax+b=0, where a, b are integers.

```
class P1{
 public static void main(String args[]){
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                       //(3)
      System.out.println("Solutie "+(-b/a));          //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
    }
}
java P1 1 1 //-1
java P1 0 3 //exception, divide by 0
            //Lines 4 or 5 are not longer executed
```

Example 1

Java VM creates the exception object corresponding to that abnormal situation and throws the exception object to those program instructions that generates the abnormal situation.

Thrown exception object can be caught or can be ignored (in our example the program P1 ignores the exception)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at P1.main(P1.java:13)
```

Catching exceptions

Using try-catch statement:

```
try{
  //code that might generates abnormal situations
}catch(TipExceptie numeVariabila){
  //treatment of the abnormal situation
}
```

Execution Flow:

If an abnormal situation occurs in the block try(), JVM creates an exception object and throws it to the block catch. If no abnormal situation occurs, try block normally executes.

If the exception object is compatible with one of the exceptions of the catch blocks then that catch block executes

Example 2

```
class P2{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));    //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }
    }
}
java P2 1 1 //Solutie -1
java P2 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
```

Multiple catch clauses

```
try{
  //code with possible errors
}catch(TipExceptie1 numeVariabila1){
  //instructions
}catch(TipExceptie2 numeVariabila2){
  //instructions
}...
catch(TipExceptien numeVariabilan){
  // instructions
}
```

Example 3

```
class P3{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));     //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P3 a b");       //(7)
     }    }
}
java P3 1 1 //Solutie -1
java P3 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
java P3 1  //java P3 a b
```

Nested try statements

```
class P4{
  public static void main(String args[]){
    try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      try{
         if (b % a==0)                 //(3)
      System.out.println("Solutie "+(-b/a));              //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
      }catch(ArithmeticException e){
          System.out.println("Nu exista solutie");  //(6)
      }
    }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P4 a b");       //(7)
     } }}
java P4 1 1 //Solutie -1
java P4 0 3 // Nu exista solutie
java P4 1  //java P4 a b
```

Nested try statements

```
try{
 //...
  try{
    //...
  }catch(TipExceptie_ii numeVar_ii){
   //...
   }
}catch(TipExceptie_1 numeVar_1){
  //instructiuni
}catch(TipExceptie_n numeVar_n){
  // ...
   try{
    //...
   }catch(TipExceptie_in numeVar_in){
   //...
   }
}
```

Finally clause

The finally clause is executed in any situation:

```
try{
  //...
}catch(TipExceptie1 numeVar1){
  //instructiuni
}[catch(TipExceptien numeVarn){
  // ...
}]
[finally{
  //instructiuni
}]
```

Finally Clause

```
A
try{
   B
}catch(TipExceptie nume){
    C
}finally{
  D
}
E
```

Block D executes:

- After A and B (before E) if no exception occurs in B. (A, B, D, E)
- After C, if an exception occurs in B and that exception is caught (A, a part of B, C, D, E).
- Before exit from the method:
  - » An exception occurs in B, but is not caught (A, a part of B, D).
  - » An exception occurs in B, it is caught but a return exists in C (A, a part of B, C, D).
  - » If a return exists in B (A, B, D).

Finally Clause

```
public void writeElem(int[] vec) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileWriter("fisier.txt"));
         for (int elem:vec)
             out.print(" "+elem);
    } catch (IOException e) {
         System.err.println("IOException: "+e);
    }finally{
         if (out != null)
             out.close();
    }
}
```

General form of Try statement

```
try{
  //code with possible errors
}[catch(TipExceptie1 e1){
  //...
}]
//...
[catch(TipExceptien en){
   //...
}]
[finally{
  //instructions
 }]
```

Defining exception classes

- By deriving from class `Exception:`

```
public class ExceptieNoua extends Exception{}

public class ExceptieNoua extends Exception{
  public ExceptieNoua(){}
  public ExceptieNoua(String mesaj){
    super(mesaj);
  }
}
```

Exceptions Specification

- Use keywordl `throws` in method signatures:

```
public class ExceptieNoua extends Exception{}

public class A{
  public void f() throws ExceptieNoua{
    //...
  }
}
```

- Many exceptions can be specified (their order does not matter):

```
public class Exceptie1 extends Exception{}
public class B{
  public int g(int d) throws ExceptieNoua, Exceptie1{
    //...
 }
}
```

Throwing exceptions

- Statement `throw :`

```
public class B{
  public int  g(int d) throws ExceptieNoua,Exceptie1{
    if (d==3)
  return 10;
    if (d==7)
        throw new ExceptieNoua();
    if (d==5)
       throw new Exceptie1();
    return 0;
  }
  //...
}
```

- Statement throw throws away the exception object and the method execution is interrupted.
- All exceptions thrown inside a method must be specified in the method signature.

Calling a method having exceptions

- use try-catch to treat the exception:

```
public class C{
  public void h(A a){
    try{
      a.f();
    }catch(ExceptieNoua e){
      System.err.println(" Exceptie "+e);
    }
  }
}
```

- Throwing away an uncaught exception (uncaught exception must be specified in the signature):

```
public class C{
  public void t(B b) throws ExceptieNoua {
    try{
      int rez=b.g(8);
    }catch(Exceptie1 e){     //only Exceptie1 is caught
      System.err.println(" Exceptie "+e);
    }
  }
}
```

Exception specification

- The subclass constructor must specified all the base class constructor (explicitly or implicitly called) in its signature.
- The subclass constructor may add new exceptions to its signature.

```
public class A{
  public A() throws Exceptie1{
  }
  public A(int i){ }
  //...
}

public class B extends A{
  public B() throws Exceptie1{  }
  public B(int i){
    super(i);
  }
  public B(char c) throws Exceptie1, ExceptieNoua{
  }
  //...
}
```

Exceptions and method overriding

- An overriding method may declare a part of the exceptions of the overridden method.
- An overriding method may add only new exceptions which are inherited from the overridden method exceptions
- The same rules are applied for the interfaces.

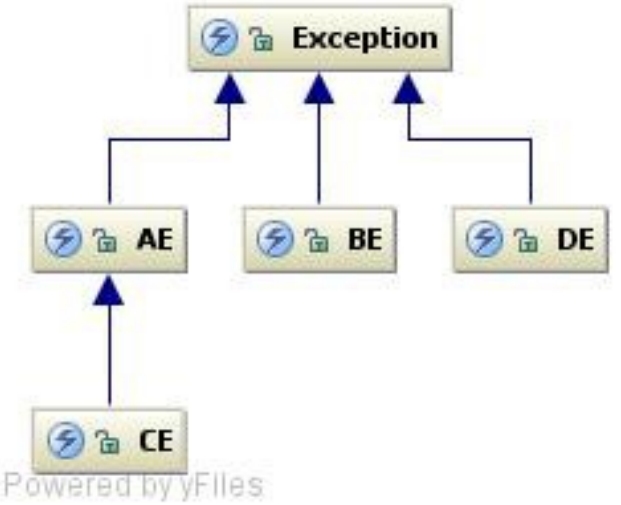
```
public class AA {
    public void f() throws Exceptie1, Exceptie2{ }
    public void g(){ }
    public void h() throws Exceptie1{  }
}

public class BB extends AA{
    public void f() throws Exceptie1{ }  //Exceptie2 is not declared
    public void g() throws Exceptie2{ }  //not allowed
    public void h() throws Exceptie3{ }
}
public class Exceptie3 extends Exceptie1{...}
```

Exceptions and method overriding

```
public class A {
    public void f() throws AE, BE {}
    public void g() throws AE{}
}

public class B extends A{
    public void g(){}
    public void f() throws AE, BE, CE{}
    public void f() throws AE, BE, DE{}
    public void g() throws DE{}
}
//?
```



Unchecked Exceptions

- Checked exceptions are those which are derived from class `Exception`
- Exceptions may be derived from class `RuntimeException`. They are named *unchecked exceptions*.

```
public class ExceptieNV extends RuntimeException{
    public ExceptieNV() { }
    public ExceptieNV(String message) {
     super(message);
    }
}
```

- Uncheked Exceptions must not be declared in the method signature.
- Unchecked Exceptions are not caught by the statement try-catch.
- Unchecked Exceptions are used only for the abnormal situations that can not be solved (the recovering cannot be done).

Exceptions order in catch clauses

- The order of catch clauses is important since the JVM selects the first catch clause on which the try block thrown exception matches.
- An exception A matches an exception B if A and B have the same class or A is a subclass of B.

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch(Exception e){...

    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    }
  }
}
```

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    } catch(Exception e){...
    }
  }
}
```

Lost exceptions

```
public class C {
    public void g(B b){
        try{
            b.f();
        }
        catch (CE ce) { }  //At least an error message must be printed
        catch (AE ae) { }
        catch (BE be) { }
    }
}
```

Re-throwing an exception

- A caught exception can be re-thrown

```
public class C {
  public int h(A a) throws Exceptie4, BE {
        try {
            a.f();
        } catch (BE be) {
            System.out.println("Exceptie rearuncata "+be);
            throw be;
        } catch (AE ae) {
            throw new Exceptie4("mesaj", ae);
        }
        return 0;
    }
}
public class Exceptie4 extends Exception {
    public Exceptie4() { }
    public Exceptie4(String message) {
        super(message);
    }
    public Exceptie4(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Exception class

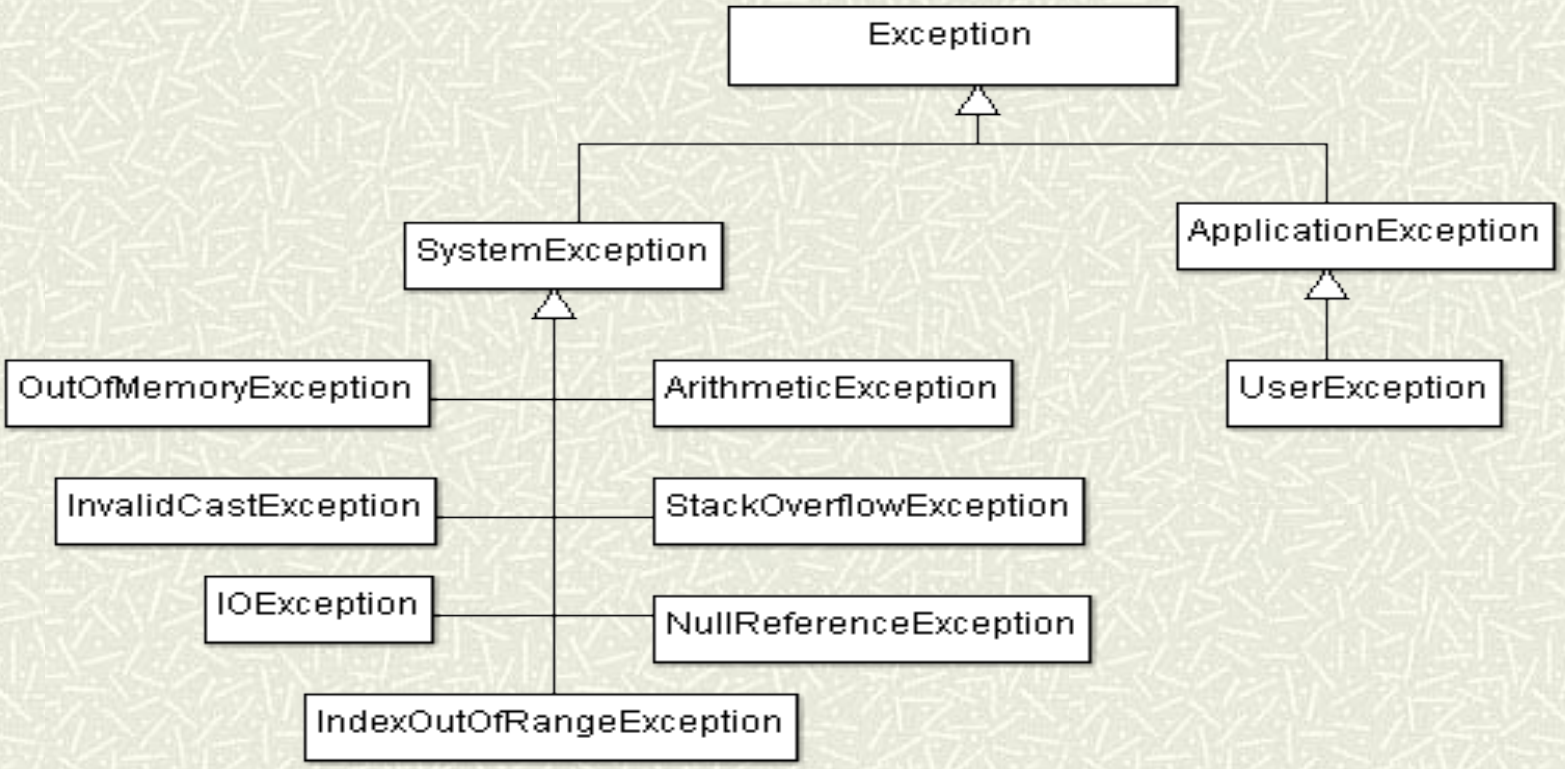
- Constructors:
  - `Exception()`
  - `Exception(String message)`
  - `Exception(String message, Throwable cause)`
  - `Exception(Throwable cause)`
- Methods:
  - `getCause(): Throwable`
  - `getMessage(): String`
  - `printStackTrace()`
  - `printStackTrace(PrintStream s)`

```
public class C {
  public int h(A a) throws BE {
         try {
             a.f();
         } catch (BE be) {
             System.out.println("Exceptie rearuncata "+be.getMessage());
             throw be;
         } catch (AE ae) {
             ae.printStackTrace();
         }
         return 0;
     }
}
```

C# EXCEPTIONS

C# Exceptions

`System` namespace



- All exceptions in C# are unchecked exceptions.
- There is no equivalent to Java's compile-time checked exceptions.

User Defined Exception

Inherits from `ApplicationException`

There are four constructors inherited from `Exception` that can be called:

- The default constructor.
- A constructor that takes a `String` as a message.
- A constructor that takes a message and an inner, lower-level `Exception`.
- `Exception(SerializationInfo, StreamingContext).`

```
class StockException : ApplicationException{
  public StockException(){ }
  public StockException(String message) : base(message) { }
  public StockException(String msg, Exception exp):base(msg,exp) { }
}
```

Try-catch block

```
try {
   // Code that might generate exceptions
} catch(Exception1 e1) {
   // Handle exceptions of type Exception1
}
// more catch block
catch(Exceptionn en) {
   // Handle exceptions of type Exceptionn
}finally{
   // code that is always executed
}
```

Example:

```
try{
   int a=10, b=0;
   int d=a/b;
}catch(Exception e){
   Console.WriteLine("Exception "+e);
}
```

Exceptions

Retrowing an exception:

```
try{
 //code
}catch(AnException e){
 throw new AnotherException("text",e);
}catch(ABException e){
 throw;
}
```

An exception can be caught without specifying a variable:

```
catch (StackOverflowException){   // no variable
   ...    //the exception methods or properties cannot be accessed
}
```

Both the variable and the type can be omitted (meaning that all exceptions will be caught):

```
catch { ... }
```

Properties of `System.Exception`

- `StackTrace` A string representing all the methods that are called from the origin of the exception to the catch block.
- `Message` A string with a description of the error.
- `InnerException` The inner exception (if any) that caused the outer exception. This, itself, may have another InnerException.

```
try{
   …
}catch(AnException e){
   Console.WriteLine(“Error message {0}“,e.Message);
   Console.WriteLine("Stack Trace {0}", e.StackTrace);
}
```

Finally clause

- A **`finally`** block is **always** executed after the **`try`** block even if no exceptions are thrown
  - It may be used to free resources
- **`return`** statements cannot occur in **`finally`** blocks
- **`goto`**, **`break`**, and **`continue`** statements can occur in**`finally`** blocks only if they do not transfer control outside the **`finally`** block itself
- The above restrictions disallow tricky cases that are allowed inJava

Valid Java code but invalid C# code:

```
public int foo() {
try { return 1; }
finally { return 2; }
}

public void foo() {
int b = 1;
while (true) {
    try { b++; throw new Exception(); }
    finally { b++; break; }
}
b++;
}
```

JAVA PACKAGES

Packages

- Groups classes and interfaces
- Name space management

- ex. package `java.lang` contains the classes `System, Integer, String`, etc.
- A package is defined by the instruction `package:`

```
//structuri/Stiva.java
package structuri;
public interface Stiva{
   //...
}
```

Obs:

1. `package` must be the first instruction of the java file
2. The file is saved in the folder `structuri` (case-sensitive).

```
//structuri/liste/Lista.java
package structuri.liste;     //folder structuri/liste/Lista.java
public interface Lista{
   //...
}
```

Packages

- Compilation:

if the file Stiva.java is in the folder

```
C:\users\maria\java\structuri
```

the current folder must be:

```
C:\users\maria\java
```

```
C:\users\maria\java> javac structuri/Stiva.java
C:\users\maria\java> javac structuri/liste/Lista.java
```

File `.class` **is saved in the same folder.**

```
C:\users\maria\java\structuri\Stiva.class
C:\users\maria\java\structuri\liste\Lista.class
```

Package

- Using the class

```
package structuri.liste;
public class TestLista{
  public static void main(String args[]){
    Lista li=...
  }
}
```

Compilation:

```
C:\users\maria\java> javac structuri/liste/TestLista.java
```

Running:

```
C:\users\maria\java> java structuri.liste.TestLista
```

Using the classes declared in the packages

```
// structuri/ArboreBinar.java
package structuri;
public class ArboreBinar{
  //...
}
```

- The classes are referred using the following syntax:

```
  [pac1.[pac2.[...]]]NumeClasa
//TestStructuri.java
public class TestStructuri{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    //...
  }
}
```

Using the classes declared in the packages

- Instruction `import`:
  - one class:

```
import pac1.[pac2.[...]]NumeClasa;
```

  - All the package classes, but not the subpackages:

```
import pac1.[pac2.[...]]*;
```

- A file may contain multiple import instructions. They must be at the beginning before any class declaration.

```
//structuri/Heap.java
package structuri;
public class Heap{
  //...
}
//Test.java
//fara instructiuni import
public class Test{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }}
```

Using the classes declared in the packages

```
//Test.java
import structuri.ArboreBinar;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }
}
//Test.java
import structuri.*;
import structuri.liste.*;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    Heap hp=new Heap();
    Lista li=new Lista();
  }
}
```

Package+import

- The instruction `package` must be before any instruction `import`

```
//algoritmi/Backtracking.java
package algoritmi;
import structuri.*;

public class Backtracking{
  //...
}
```

- The package `java.lang` is implicitly imported by the compiler.

Static import

- Starting with version 1.5

```
import static pac1.[pac2.[. ...]]NumeClasa.MembruStatic;
import static pac1.[pac2.[...]]NumeClasa.*;
```

- Allow to use static members of class `NumeClasa` without using the class name.

```
package utile;
public class EncodeUtils {
    public static String encode(String txt){...}
    public static String decode(String txt){...}
}
//Test.java
import static utile.EncodeUtils.*;
public class Test {
    public static void main(String[] args) {
        String txt="aaa";
        String enct=encode(txt);
        String dect=decode(enct);
        //...
    }
}
```

Anonymous package

```
//Persoana.java
public class Persoana{...}

//Complex.java
class Complex{...}

//Test.java
public class Test{
  public static void main(String args[]){
    Persoana p=new Persoana();
    Complex c=new Complex();
    //...
  }
}
```

If a file `.java` does not contain the instruction `package`, all the file classes are part of anonymous package.

Name Collision

```
// unu/A.java
package unu;
public class A{
  //...
}
```

```
// doi/A.java
package doi;
public class A{
  //...
}
```

```
//Test.java
import unu.*;
import doi.*;
public class Test{
  public static void main(String[] args){
    A a=new A();  //compilation error
    unu.A a1=new unu.A();
    doi.A a2=new doi.A();
  }
}
```

Access modifiers

- 4 modifiiers for the class members:
  - `public:` access from everywhere
  - `protected:` access from the same package and from subclasses
  - `private:` access only from the same class
  - `:` access only from the same package
- Classes (excepting inner classes) and interfaces can be public or nothing.

Access modifiers

```
// structuri/Nod.java
package structuri;
class Nod{
   private Nod urm;
   public Nod getUrm(){...}
   void setUrm(Nod p){...}
  //...
}
```

```
// structuri/Coada.java
package structuri;
public class Coada{
   Nod cap;
   public Coada(){ cap.urm=null;}
   Coada(int i){...}
  //...
}
```

```
//Test.java
import structuri.*;
class Test{
  public static void main(String args[]){
      Coada c=new Coada();
      Nod n=new Nod();     //class is not public
      Coada c2=new Coada(2);   //constructor is not public
 }
}
```

Access modifiers

```
package unu;
public class A{
  A(int c, int d){...}
  protected A(int c){...}
  public A(){...}
  protected void f(){...}
  void h(){...}
}
```

```
package unu;
class DA extends A{
   DA(int c){ super(c);}
}
```

```
package doi;
import unu.*;
class DDA extends A{
  DDA(int c){super(c);}
  DDA(int c, int d) {super(c,d);}
  protected void f(){
    super.h();
  }
}
```

Core packages in Java (6.0)

java.lang (basic language functionality, fundamental types, automatically imported)

java.util (collections and data structures)

java.io and **java.nio** (old/new file operations API)

java.math (multi-precision arithmetic)

java.net (networking, sockets, DNS lookup)

java.security (cryptography)

java.sql (database access: JDBC)

java.awt (native GUI components)

javax.swing (platform independent rich GUI components)

C# Namespace

Namespaces

Classes can be grouped in namespaces

- A hierarchical grouping of classes and other entities
- Every source file defines a global namespace
  - possibly implicitly, if the user doesn't provide a name
- Affects visibility of various classes

Unlike Java, there need not be any connection between namespaces and directory structure

The following are allowed in C# and disallowed in (the official implementation of) Java:

- multiple public classes in the same file
- splitting the declaration of a class across multiple files

The Global Namespace

The global namespace consists of:

All top-level namespaces

All types not declared in any namespace

```
class Utils  {}
namespace tests {
  namespace model  {
        class Person {…}
        class Question{…}
    }
}
```

The class `Utils` and the namespace `tests` belong to the global namespace.

Namespaces

Remarks

All names present in outer namespaces are implicitly imported into inner namespaces.

If you want to refer to a type in a different branch of your namespace hierarchy, you can use a partially qualified name.

```
namespace tests{
 namespace model{
  class Person{}
 }
 namespace gui{
  class TestForm {
     model.Person p;
  }
 }
}
```

Names in inner namespaces hide names in outer namespaces.

Namespaces

Remarks

All names present in outer namespaces are implicitly imported into inner namespaces.

If you want to refer to a type in a different branch of your namespace hierarchy, you can use a partially qualified name.

```
namespace tests{
 namespace model{
  class Person{}
 }
 namespace gui{
  class TestForm {
     model.Person p;
  }
 }
}
```

Names in inner namespaces hide names in outer namespaces.

Namespaces

A namespace declaration can be repeated, as long as the type names within the namespaces do not conflict.

```
namespace tests.model{
  class Person{}
}
namespace tests.model{
  class Question{}
}
```

The `using` directive can be nested within namespaces.

```
namespace N1 {
  class Class1 {}
}
namespace N2{
  using N1;
  class Class2 : Class1 {}
}
namespace N2 {
  class Class3 : Class1 {}   // compile error
}
```

Aliasing Types and Namespaces

The `using` directive can be used for declaring an alias for a type or for a namespace:

```
using person=tests.model.Person;   //alias for a type
using win=tests.gui;         //alias for a namespace
class Test{
 void Main(){
   person p=new person();
   win.TestForm tf=new win.TestForm();
 }
}
```

- `object` is an alias for `System.Object`
- `string` is an alias for `System.String`

BCL (Base Class Library)

System

(basic language functionality, fundamental types)

System.Collections (collections of data structures)

System.IO (streams and files)

System.Net (networking and sockets)

System.Reflection (reflection)

System.Security

(cryptography and management of permissions)

System.Threading (multithreading)

System.Windows.Forms

(GUI components, nonstandard, specific to the Windows platform)