Secure Applications Programming JSE & JCA

Lect. Catalin Boja, Ph.D.
IT&C Security Master
catalin.boja@ie.ase.ro

Course organization

- **Activities:** Course 50% + Laboratory 50%
- **Language:** English
- **Evaluation:** Written Quiz Exam on E-Evaluation platform
- **Objective:** Gaining theoretical and practical knowledge needed to develop Java applications and to implement cryptographic solutions using JCA – Java Cryptography Architecture

Bibliography

1. Ion IVAN, Cristian TOMA – *Informatics Security Handbook*, 2nd Edition, ASE Printing House, 2010
2. Cristian TOMA – *Security in Software Distributed Systems*, ASE Printing House, 2008
3. Jonathan Knudsen, Patrick Niemeyer - *Learning Java*, 3rd Edition, O'Reilly
4. David Hook - *Beginning Cryptography in Java*, Wrox Press
5. http://java.sun.com
6. www.wikipedia.com / www.google.com

JSE

Section I – Java fundamentals

- basic concepts JDK, JRE, JVM
- Prerequisites
- Using an IDE – NetBeans / Eclipse
- Basic OOP Concepts: data types, arrays, class, object, reference, cloning mechanism, exceptions, String, immutable, derivation, interface, abstract class, polymorphism, Late-binding and virtual methods

JSE

Section II – Java Advanced Topics

- Serialization
- Java Generics & Java Add-notations
- Java Collection Framework
- Threads
- I/O Stream (Files and Network) + Java Libraries
- Design patterns: Factory, Singleton
- JNI – Java Native Interface

Cryptography Fundamentals

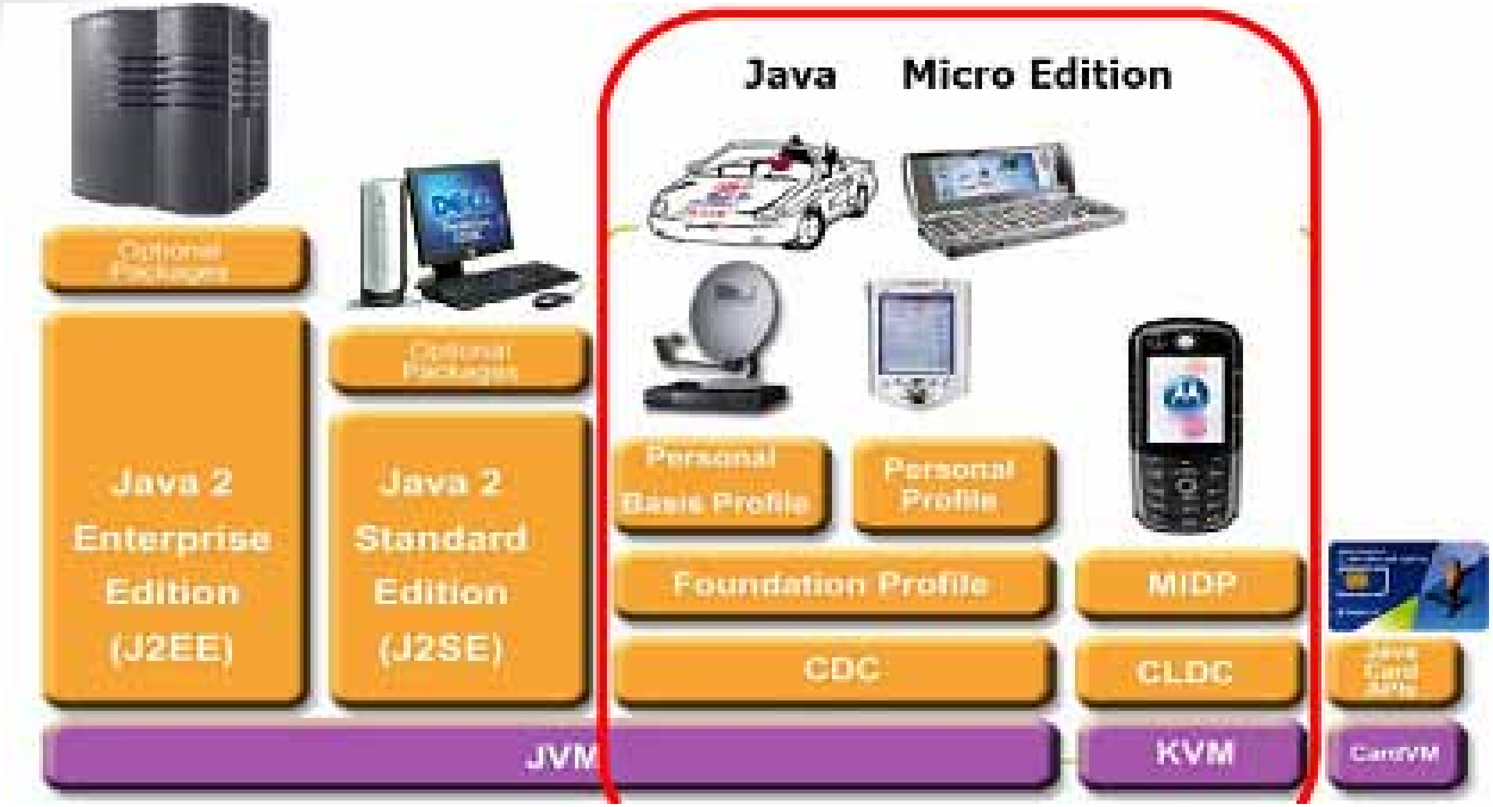
Section III – JCA – Java Cryptography Architecture

- Hash functions – MD5, SHA-1
- Symmetric encryption– DES/AES in ECB and CBC mode
- Asymmetric encryption– RSA
- Digital certificates – X509 v3 (using *keytool* tool or a source code application)

Section IV – Java GUI

- Java Swing

Java


Java Micro Edition
Optional Packages
Optional Packages
Java 2 Enterprise Edition (J2EE)
Java 2 Standard Edition (J2SE)
Personal Basis Profile
Personal Profile
Foundation Profile
MIDP
CDC
CLDC
Java Card APIs
JVM
KVM
CardVM

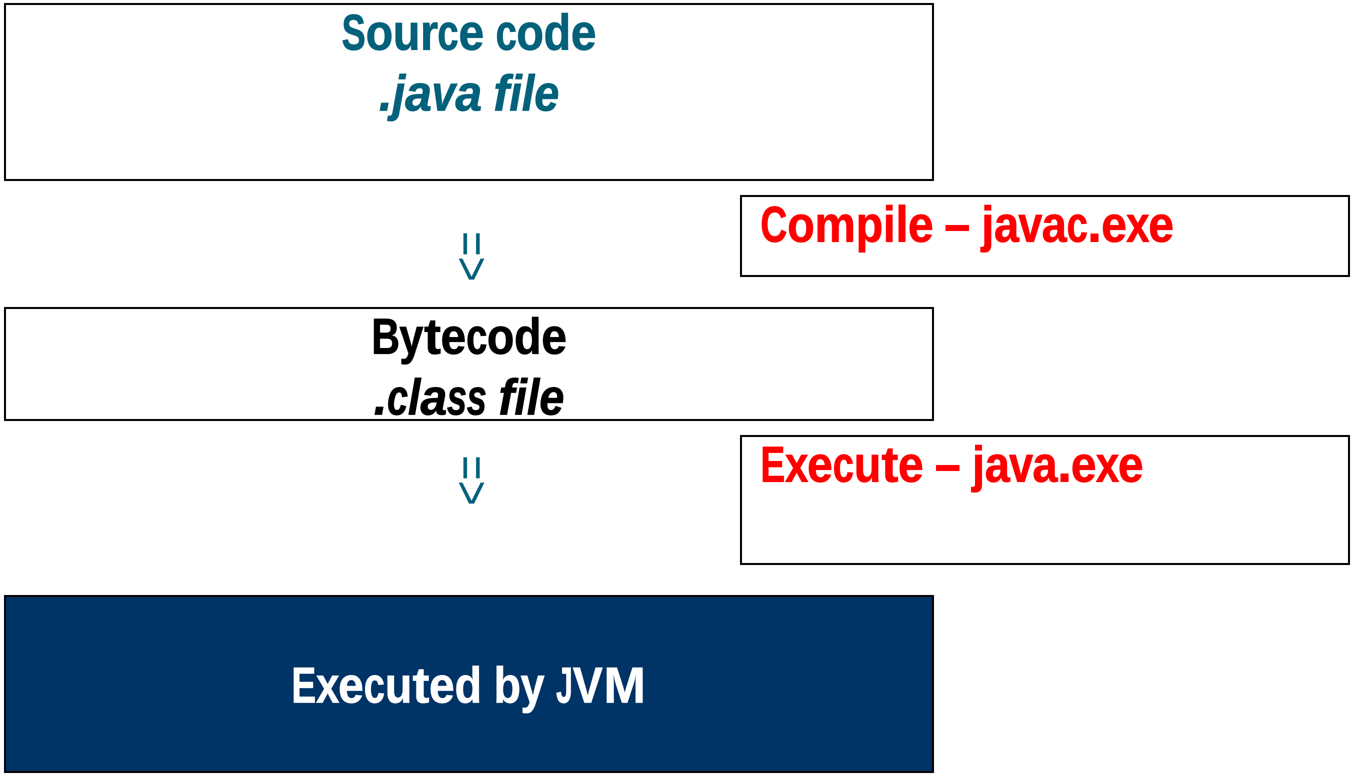
Java fundamentals

- JDK, JRE, JVM
- Comparing with C/C++ applications
- Advantages and Disavantages

Java fundamentals

- Java Application Structure
- Java Development Steps
- Hello World application
- Command line development with JDK 6.0
- Using an IDE - NetBeans
- **\Laboratory\P01_JSE\S01_Hello**

Java fundamentals



Source code

.java file

⇓

Compile – javac.exe

Bytecode

.class file

⇓

Execute – java.exe

Executed by JVM

http://www.itcsolutions.eu/2010/11/29/tutorial-java-1-prerequisites/

Java fundamentals

1. Needed software and tools

- a simple ASCII editor (Notepad, Notepad++, JEdit, or other) for writing source files;
- Java compiler, *javac.exe* to compile source code files, *. java*, and to obtain bytecode files with *.class* extension;
- virtual machine (Java Virtual Machine – JVM), *java.exe* to run Java applications

Java fundamentals

- *javac.exe* and *java.exe* are obtained by installing the Java SDK (Software Development Kit), JDK which is obtained from the java.sun.com site;
- the two executables are available in **C:\Program Files\Java\jdk1.6.0_16\bin** if you chose the default location;

Java fundamentals

1. Edit the source code;
2. Open the command prompt; select **Start**, type **cmd.exe** and press **Enter**;
3. Check if the system knows where to find the two executables, *java.exe* and *javac.exe*
4. Set environment variables (if needed)

```
set JAVA_HOME=C:\Program Files\Java\jdk1.6.0_16
set PATH=%JAVA_HOME%\bin
set CLASSPATH=%JAVA_HOME%\jre\lib;
```

Java fundamentals

5. Compile the source code

```
d:\Java Workspace> javac.exe HelloWorld.java
```

6. Run the Java application

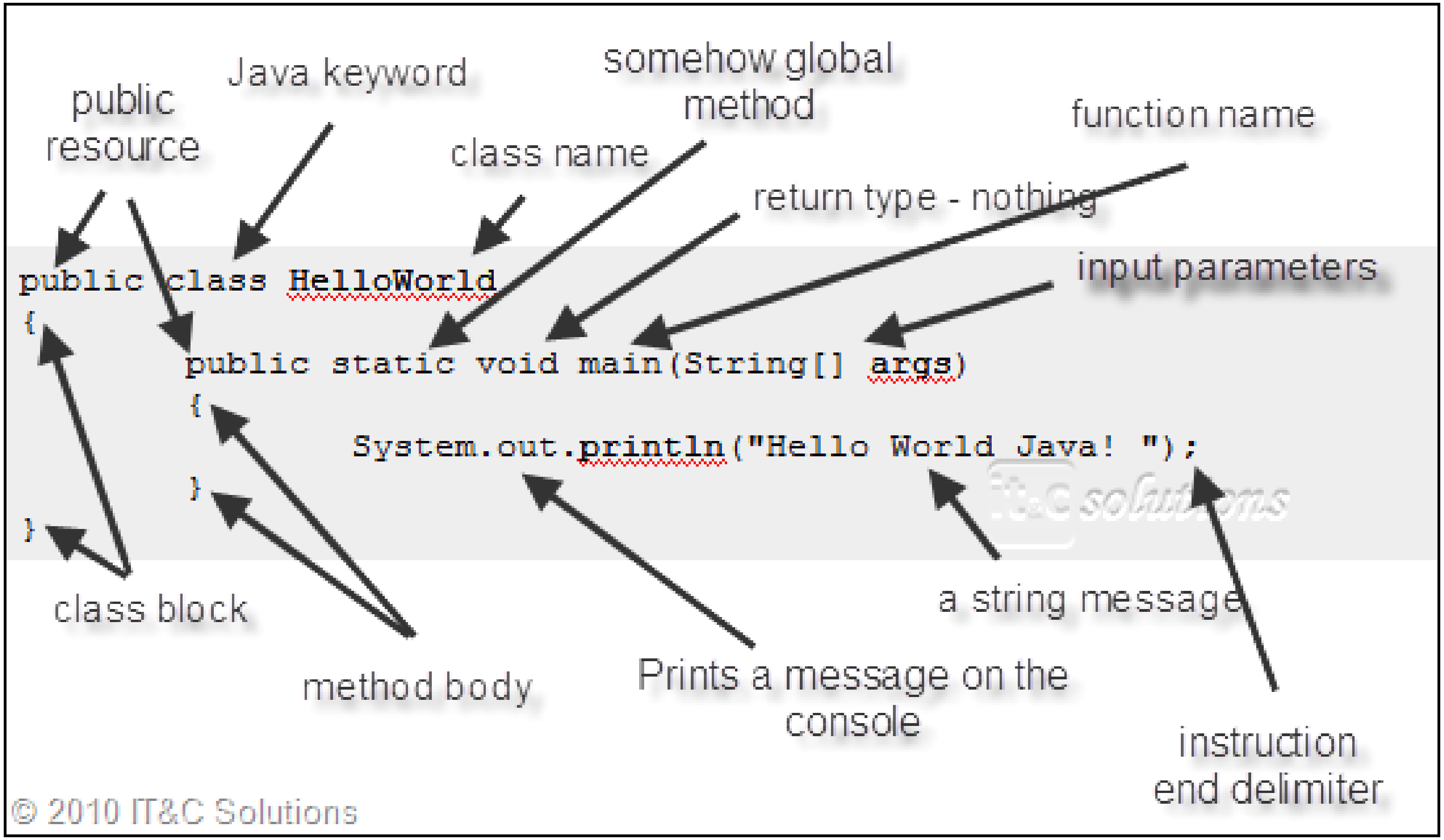
```
d:\Java Workspace> java.exe HelloWorld
```

An alternative is to use an IDE:

- NetBeans, available at http://www.netbeans.com/ (Java SE)
- Eclipse, available at http://www.eclipse.org/ (Eclipse IDE for Java Developers version)

Java fundamentals

HelloWorld.java file



```
public class HelloWorld
{
          public static void main(String[] args)
          {
                    System.out.println("Hello World Java! ");
          }
}
```

http://www.itcsolutions.eu/2010/12/15/tutorial-java-2-basic-concepts/

Java fundamentals

- one line comments defined by //
- multiple lines comments defined between /* and */
- the end delimiter for a instruction is ; (semicolon);
- commented instructions are ignored;
- instructions can be associated in blocks of code that are defined between { and };
- Java language is case sensitive, **vb** as variable is different than **Vb** or **VB**;

Java fundamentals

- EVERYTHING is defined inside a class
- You CAN NOT define GLOBAL variables or methods outside a class (like in C/C++);
- a class has a body defined between { and };
- The class that contains the main function has the same name (at case level) with the file;

Java fundamentals

How to use an IDE:

- NetBeans, available at http://www.netbeans.com/ (Java SE)
- Eclipse, available at http://www.eclipse.org/ (Eclipse IDE for Java Developers version)

Java fundamentals

Object Oriented Programming concepts in Java:

In Java all objects are managed only by references.

Java fundamentals
C++ vs. Java

C++

```
class Student{ … };
void main(){

Student a1(2345, "Maria");
Student a2(231, "Ana");
a2 = a1;

Student* pa1 = new Student(112, "Ion");
Student* pa2 = new Student(128, "Vasile");

pa1 = pa2;

pa1->print();
pa2->print();

if(a1 == a2) cout << "\n\t equal VALUES";
if(pa1 == pa2) cout<< "\n\t equal POINTERS";
   else cout<< "\n\t NOT equal POINTERS"
}
```

Java

```
using System;
public class Student{
                …
public static void main(String[] args){

Student a1 = new Student(2345, "Maria");
Student a2 = new Student(231, "Ana");

a2 = a1;

a1.print();
a2.print();

if (a1 == a2)  System.out.println("\t equal REFERENCES");
     else System.out.println("\n NOT equal REFERENCES");         }
}
```

Java fundamentals
C++ vs. Java

- objects managed by value and reference;
- a class may contain dynamic attributes managed by pointers;
- the destructor is used to release memory space and to avoid memory leaks;
- you must define copy constructor and overload = operator to prevent default shallow-copy
- strings managed by char *

- **objects are managed only by references**
- **pointers from C++ defined with * can be used only in native code – JNI;**
- **memory clean-up is done by the JVM garbage collector;**
- **a destructor like method (finalize) used to clean up other resources**
- **you CAN'T overload operators;**
- **operator = does ALLWAYS shallow copy ;**
- **copy constructor needed to make deep copy;**
- **strings managed by String (object used like a vaue-type);**

Java fundamentals - variables

Variables in Java:

1. Primitive data types
   - Integer, boolean, floating point values, char [How to define primitive data types variables]
2. References
   - Objects
   - Interfaces
   - Wrappers for primitive types
   - Enums

Java fundamentals - variables

Value data type	Size	Range for signed values	Category
byte	**1 byte**	-128 –> 127	integer
short	**2 bytes**	-32768 –> 32767	integer
int	**4 bytes**	-2147483648 –> 2147483647	integer
long	**8 bytes**	-9,223,372,036,854,775,808 –> 9,223,372,036,854,775,807	integer
float	**4 bytes**	7 significant digits	real simple precision
double	**8 bytes**	15 significant digits	real double precision
char	**2 bytes**	’\u0000’ –> ’\uffff’ 0 –> 65535	16 bits Unicode char
boolean	**1 bit**	true or false	logic value

For more information check How to define primitive data types variables **post**

Java fundamentals - variables

- variable name must begin with a letter, underscore symbol (_) or dollar sign ($);
- variable names can not begin with a digit;
- after first character, you can use digits in the variable name;
- variable name can not be a word reserved for Java language, a keyword;
- several variables can be defined simultaneously;

Java fundamentals - variables

- variable names are chosen by the programmer, but for efficiency, there are some naming conventions about variable names: Hungarian notation , CamelCase;

```
int iBooksNumber; //Hungarian Notation
int BooksNumber; //CamelCase
int booksNumber; //Java mixed case
```

Java fundamentals - variables

- Java is strong type language;

```
float vb2 = 23.5; //compilation error -
  possible loss of precision
int vb3 = 45.6; //compilation error -
  possible loss of precision
boolean test = 23; //compilation error -
  incompatible types
```

Java fundamentals - variables

- the type of value must be the same as the variable type;
- several variables can be initialized at the same time;
- in Java, the only possible values for Boolean variables are true or false;
- float constant values are defined with the symbol **f** in the end;
- character symbols are defined between ‘ ‘ (apostrophe) and not between " " (quotation marks);
- real values can be defined in scientific format, for example, 1.234e2 is equivalent to 123.4 ;
- integer values in base 8 are prefixed with 0, eg 021 is 17 in base 10;
- integer values in base 16 (hexadecimal representation) are prefixed with 0x, for example 0×11 is 17 in base 10;

Java fundamentals - variables

char variables can have as values a series of special characters, escape sequences:

Escape sequences	Value
\b	backspace
\t	tab
\n	line feed
\f	form feed
\r	carriage return
\”	double quotes
\’	apostrophe
\\	backslash

Java fundamentals - variables

Default values in Java (NOT for local variables; ONLY for instance variables):

Type	Default value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	‘\u0000’
boolean	false
reference	null

Java fundamentals - variables

```
public static void main()
{
int sum;
//local variable declared in main method
sum = sum + 10;
//compiler error
//variable sum might not have been
  initialized
}
```

Java fundamentals – Stack & Heap

Stack:

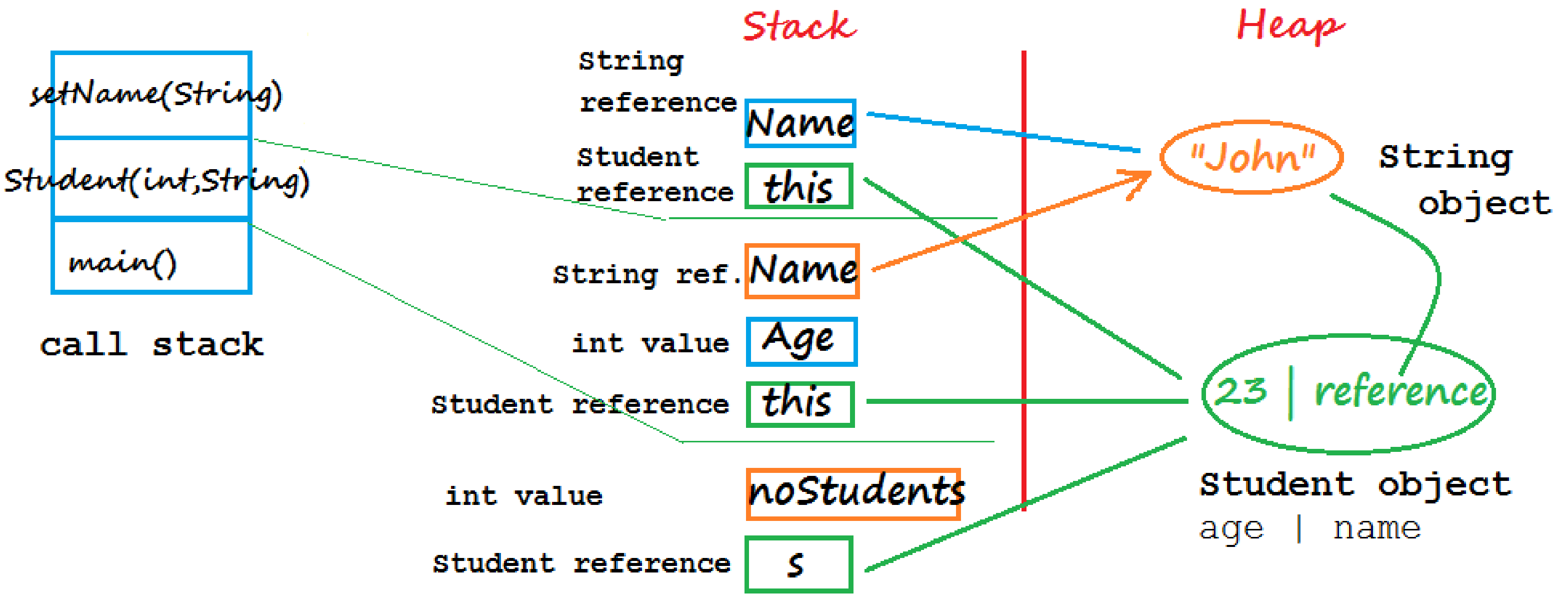
- a memory space reserved for your process by the OS;
- it is important to establish that the stack is limited and its size is fixed;
- most of the time, the stack it is used to store functions/methods variables (input arguments and local variables).
- each method has its own stack (a zone in the process stack), including **main**, which is also a function.
- a method stack exists only during the lifetime of that method: from the calling moment until the return moment;

Java fundamentals – Stack & Heap

Heap:

- a memory space managed by the OS and used by processes to get additional space at run-time;
- this area it is a global, meaning that any process can use it (of course, processes can’t read or write in another process Heap reserved area);
- the role of this memory is to provide additional memory resources to processes that need that supplementary space at run-time;
- the space needed at run-time by a process is determined by functions like *new* which are used to get additional space in Heap.

Java fundamentals – Stack & Heap



Java fundamentals – Garbage collector

- solve some problems regarding writing Java applications that will run out of memory – memory leaks in C++;
- it is a complex and very efficient JVM routine that will not interfere with your Java process performance;
- it will not save any out of memory situation
- you can request garbage collector to make a memory clean explicitly by invoking the **System.gc( )** method

Java fundamentals – Garbage collector

- it is not recommended to interfere with the garbage collector by calling **System.gc( )** method, because you have not any guarantees on how it will behave;
- In order to generate unreachable objects or memory leaks and to enjoy the benefits of having a Garbage Collector you must loose or remove all the references for that object:
  - **null a reference;**
  - **reassigning the reference**;
  - **isolating a reference**;

Java fundamentals – References

- reference variables, or references, are variables (like a primitive data type, let’s say *int vb*) because they require memory space to store their values;
- the main difference between a reference and primitive variable is that the values for the first one are numbers that represent addresses, mainly, of memory zones in Heap

Java fundamentals – References

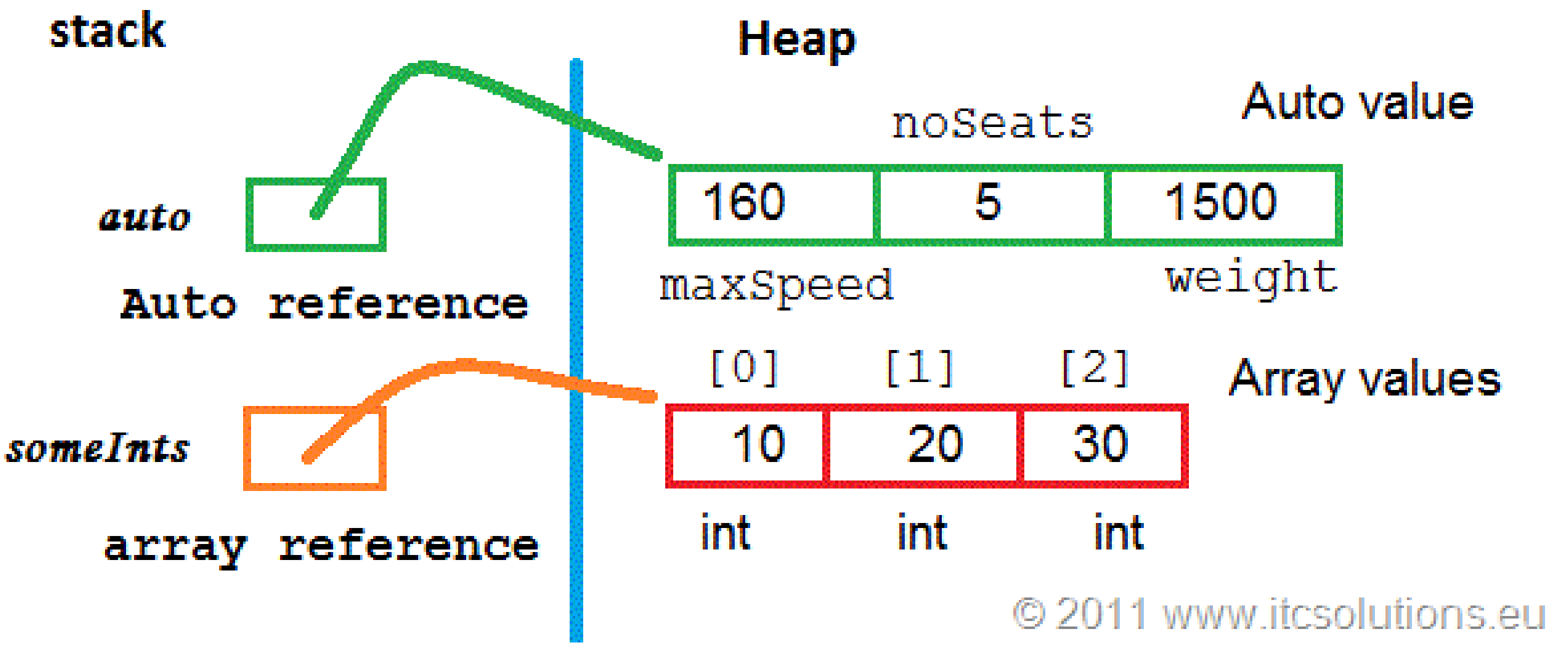
How to define reference data type variables:

class_name reference_name;

How to initialize reference data type variables:

- In order to get an address we must request and reserve some memory in Heap (only here)
- this is done with the *new* operator

Java fundamentals – References



Java fundamentals – Variable scope

- **Instance variables (attributes)** – these variables are part of an object, so, they are created when the object is created; they exists until the object is created; the object and its methods have access to its instance variables;
- **Static variables** – these variables are part of a class; they are created when the class is loaded by the JVM;

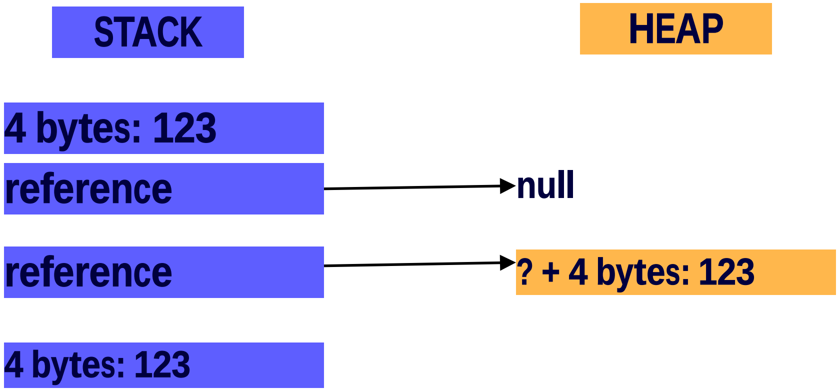
Java fundamentals – Variable scope

- **Methods local variables** – these variables are defined on the method stack and they exists as long as the method is executed (it is placed on the call stack); even the local variables can be accessed, you can't use them from a nested method (Tutorial Java – #8 Understand Stack and Heap);
- **Block variables** – these variables are defined inside blocks of code (between { and }) and can be used while the block is executed; typical blocks of code are for, while, initialization block.

Java fundamentals - Boxing

Converting a value type into a reference type and backwards is done by **boxing / unboxing**

```
public class BoxUnbox
{
        static void main(String[] args)
        {
          int i = 123;
          Integer iObject;
          iObject = i;
          int j = iObject;
        }
}
```



STACK

HEAP

4 bytes: 123

reference

null

reference

? + 4 bytes: 123

4 bytes: 123

Java fundamentals - Boxing

Value data type	Wrapper Class	Constructor Arguments
byte	Byte	byte or String
short	Short	short or String
int	Integer	int or String
long	Long	long or String
float	Float	float, double or String
double	Double	double or String
char	Character	char
boolean	Boolean	boolean or String

Java fundamentals – String and Immutable

- String
  - – objects
  - – used as primitives
  - – “String constant pool”
- Immutable
  - – objects that DO NOT change their value
  - – String, Integer + other wrappers

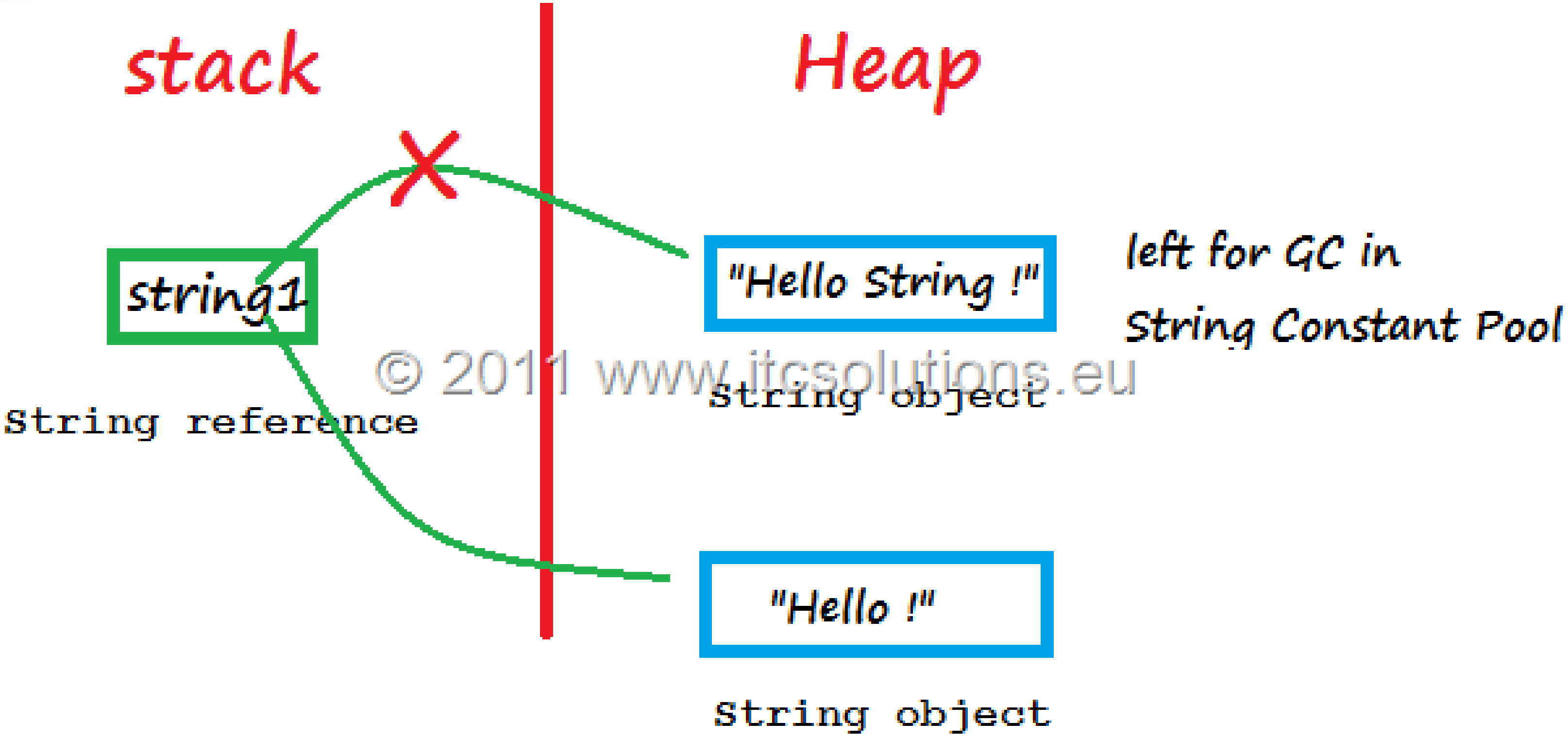
Java fundamentals – String and Immutable

- in Java every char is a 16 bit Unicode value, and not 1 byte;
- in Java, strings values are managed by String objects;
- in Java, the syntax allows you to use Strings as primitive data types (you can use = operator to initialize them);
- in Java, Strings are immutable objects, meaning that once are created, they can't change their value.

Java fundamentals – String and Immutable

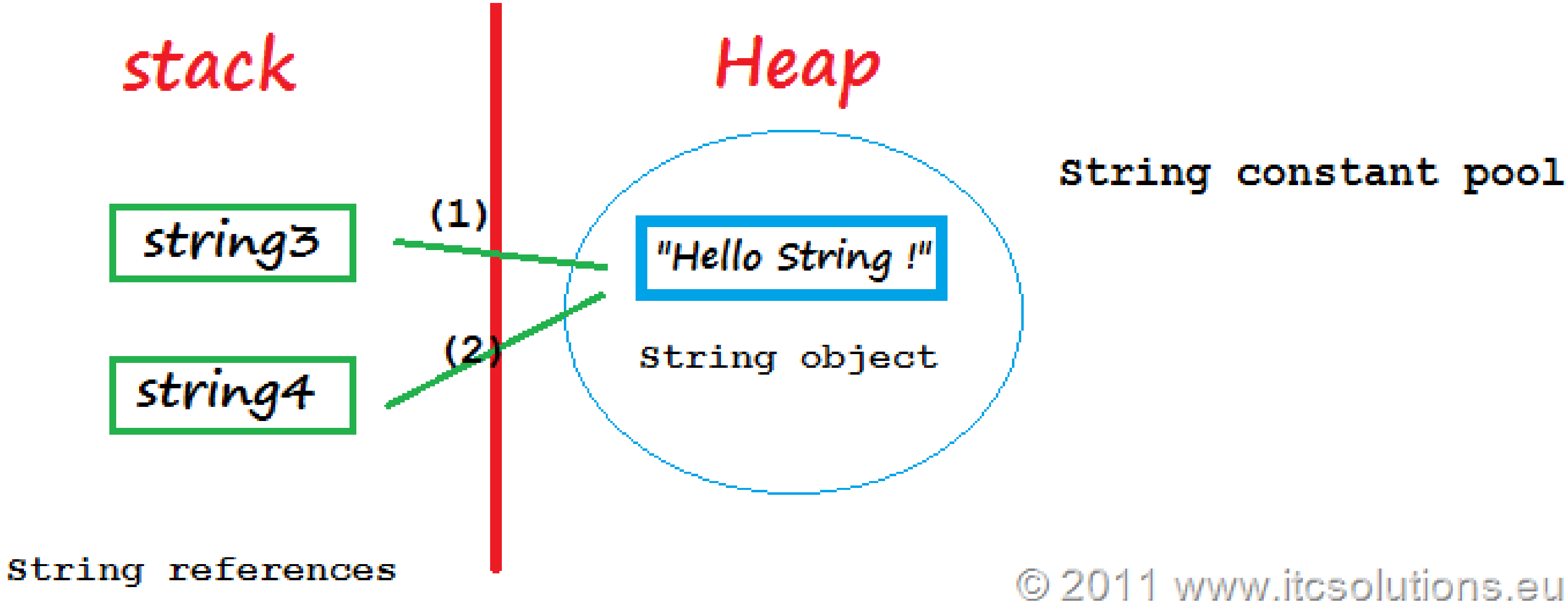
- using = between 2 String references you will copy the reference value and not the object value;
- In Java, String objects are very special (once because they are immutable) because their values are treated in a special way. For an efficient use of memory, JVM manages String values (especially String literals) by putting them in a special area of memory called the “String constant pool”.

Java fundamentals – String and Immutable



String is Immutable

Java fundamentals – String and Immutable



String Constant Pool

Java fundamentals – String and Immutable

Method	Description
charAt()	returns the char at a given index; index takes values from 0 to length()-1;
concat()	appends a String to the end of another; the same as +
equals()	compare at case level 2 String values
length()	return the number of chars; IT IS NOT the length attribute of an array. IT IS A METHOD
replace()	replace occurences of a char with a given one
substring()	returns a substring
toLowerCase()	converts all chars to lowercase
toString()	returns the value of the String object
toUpperCase()	converts all chars to uppercase
trim()	remove whitespace from the end

These methods will NOT affect the current object.

Java fundamentals – String and Immutable

StringBuilder and StringBuffer:

- these two classes are almost the same;
- provide the means for efficient I/O operations with large streams;
- their values are not stored in String Constant Pool and they behave like normal objects

Java fundamentals – String and Immutable

Method	Description
append()	adds the argument to the end of the current object
delete()	deletes chars between a start and end index
insert()	inserts a value at a given offset
reverse()	reverse the value of the current object
toString()	returns the value of the StringBuilder or StringBuffer object

These method affects the value of the calling object, StringBuilder or StringBuffer.

Java fundamentals – Operators

- Assignment operator: =
- Compound assignment operators: +=, -=, *=, /=
- Relational operators: <, <=, >, >=, ==, !=
  - The result is a boolean value (true or false)
  - The equality operators used between references are comparing the references values (objects addresses) and not the objects

Java fundamentals – Operators

- Instanceof operator
  - – Used for references to check the object type
- Arithmetic operators: +, -, *, /
- Reminder operator: %
- String concatenation operator: +
- Increment and Decrement operators: ++, --
  - – Have 2 forms: prefix and postfix

Java fundamentals – Operators

- Conditional operator:

```
(condition) ? value_for_true : value_for_false
```

- Logical operators: &, |, ^, &&, ||

Java fundamentals – Control structures

Flow control structures:

- if-then
- if-then-else
- do-while
- while-do
- for
- enhanced-for
- switch

Flow control statements

Java fundamentals – Control structures

IF-THEN

```
if (condition)
{
    < statement 1 >
    < statement 2 >
}
```

- *condition* is a boolean expression or variable that has the value *true* or *false* .For example *30*> *10* or 10 == 30

- are not accepted conditions based on expressions or variables that have numeric values

Java fundamentals – Control structures

IF-THEN-ELSE

```
if (condition)
{
    < statement 1 >
    < statement 2 >
}
else
{
    < statement 1 >
    < statement 2 >
}
```

- an alternative is the conditional operator:

condition ? then_statement : else_statement

Java fundamentals – Control structures

DO-WHILE

```
do
{
    < statement 1>
    < statement 2>
} while (condition)
```

-the condition for exiting/staying in the loop is checked at the end of the loop block

- the structure will run at least once the iteration statements

Java fundamentals – Control structures

WHILE-DO

```
while (condition)
{
    < statement 1>
    < statement 2>
}
```

- the condition for exiting from/staying in the loop is checked before the first statement in the loop block is executed

Java fundamentals – Control structures

FOR

```
for(initialization; condition;
   iteration)
{
   < statement 1>
   < statement 2>
}
```

- like *do-while;*
- is more efficient, simply because the iteration and the initialization statements are included in the structure and not in the block;

Java fundamentals – Control structures

FOR

- you can be write multiple iteration and initialization statements separated by , (comma);
- initialization, iteration and condition, are optional;

```
for( intialization; ; )
for( ; condition; iteration )
for( ; ; iteration)
for( ; ; ) // endless loop
```

Java fundamentals – Control structures

ENHANCED-FOR

```
for ( variable: iterable_collection ){
    < statement 1>
    < statement 2>
}
```

- used to iterate through a collection that implements the *java.lang.Iterable* interface

Java fundamentals – Control structures

SWITCH

```
switch (testValue) {
case constant1:
    <statement 1>
    break;
case constant2:
    < statement 2>
    break;
...
default:
     < statement >
```

- conditional structure with multiple branches;
- each *case* clause is closed with the *break* instruction because it provides the exit from the *switch* structure;
- break is optional;

Java fundamentals – Control structures

break **and** ***continue*** **statements:**

- *break* statement stops the current loop block implemented by *for, do-while*, *while-do;*
- *break* in a *switch* structure, will close the last *case* clause;
- *continue* instruction will suspend the current iteration of a loop block, *for, do-while*, *while-do* and will execute the next iteration

Java fundamentals - Arrays

SINTAX:

```
base_type array_name[];            // style similar to C/C++
base_type [ ] array_name;
```

- different than C/C++ arrays
- an instance of the *array* class – it is an object
- has methods:
  - clone( );
  - equals();
- has attributes:
  - length
- Arrays tutorial

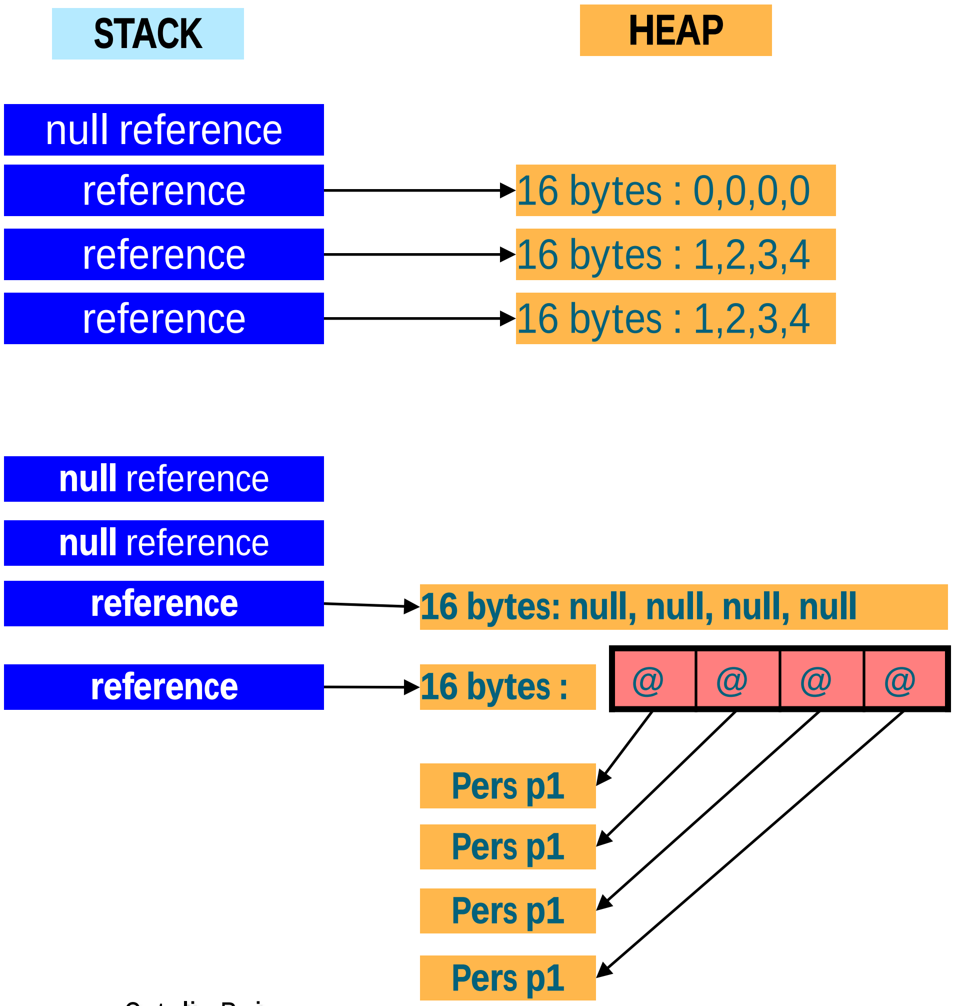
Java fundamentals - Arrays

- because they are objects, the array values are stored in HEAP;
- an array is initialized in 3 steps:
  - **Define the array;**
  - **Reserve space for it;**
  - **Initialize items value (**optional, because during the memory allocation, the elements get default values associated with the **base type** of the array)
- access to array elements is done using operator []

Java fundamentals - Arrays

```
int [ ] vect;
vect = new int[4];
int [ ] vect2 = {1,2,3,4};
int [ ] vect3 = new int[] {1,2,3,4};
```

```
Pers p1;
Pers [ ] vectPers;
vectPers = new Pers[4];
vectPers = new Pers[4] {p1,p2,p3,p4};
```



STACK

HEAP

null reference

reference → 16 bytes : 0,0,0,0

reference → 16 bytes : 1,2,3,4

reference → 16 bytes : 1,2,3,4

null reference

null reference

reference → 16 bytes: null, null, null, null

reference → 16 bytes : @ @ @ @

Pers p1

Pers p1

Pers p1

Pers p1

Java fundamentals - Arrays

- How to add a new element to a Java array

In this post we see what is the solution to the problem of adding a new element to an existing array.

- How to copy values of an array into another array

In this post are described methods used to copy one array values into another array.

- Matrixes and Multidimensional Arrays

In this post we will see what are and how to define multidimensional arrays. The most common multidimensional array is the matrix – a two dimension array.

Java fundamentals
Two-dimensional Arrays

SINTAX:

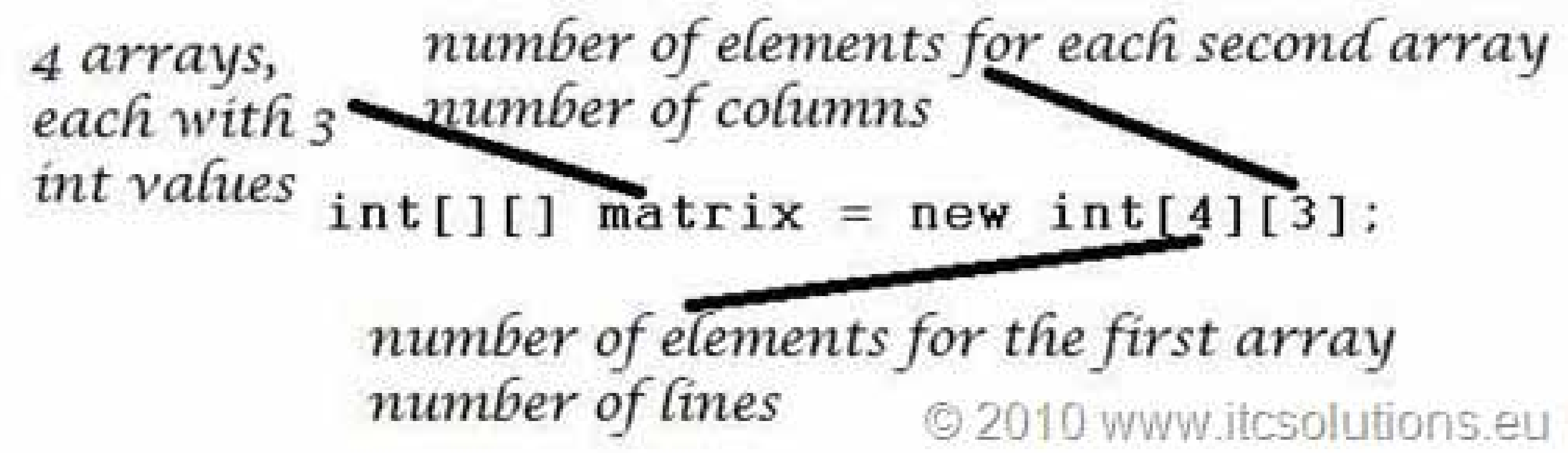
```
base_type[][] matrix_name;
base_type matrix_name[][]; // style similar to C/C++
```

```
int [][] matrix;                          //matrix
matrix = new int[3][3];

int [][] matrix = new int[3][];           //zig-zag matrix

matrix[0] = new int[3];

matrix[1] = new int[5];

matrix[2] = new int[7];
```

Matrixes and Multidimensional Arrays for more info.

Java fundamentals
Two-dimensional Arrays

- a matrix is in Java an array of arrays

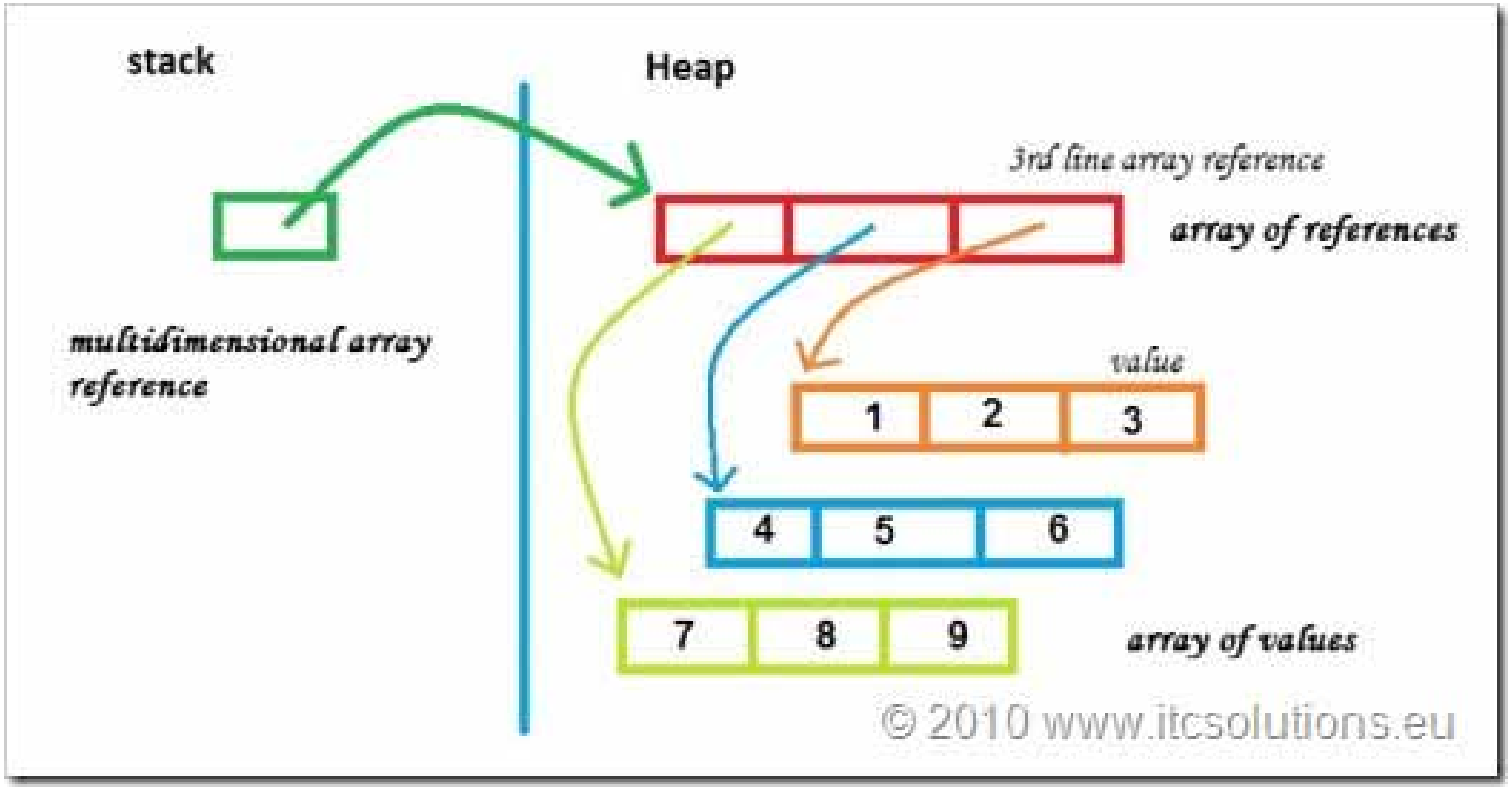


```
int[][] matrix = new int[4][3];
```

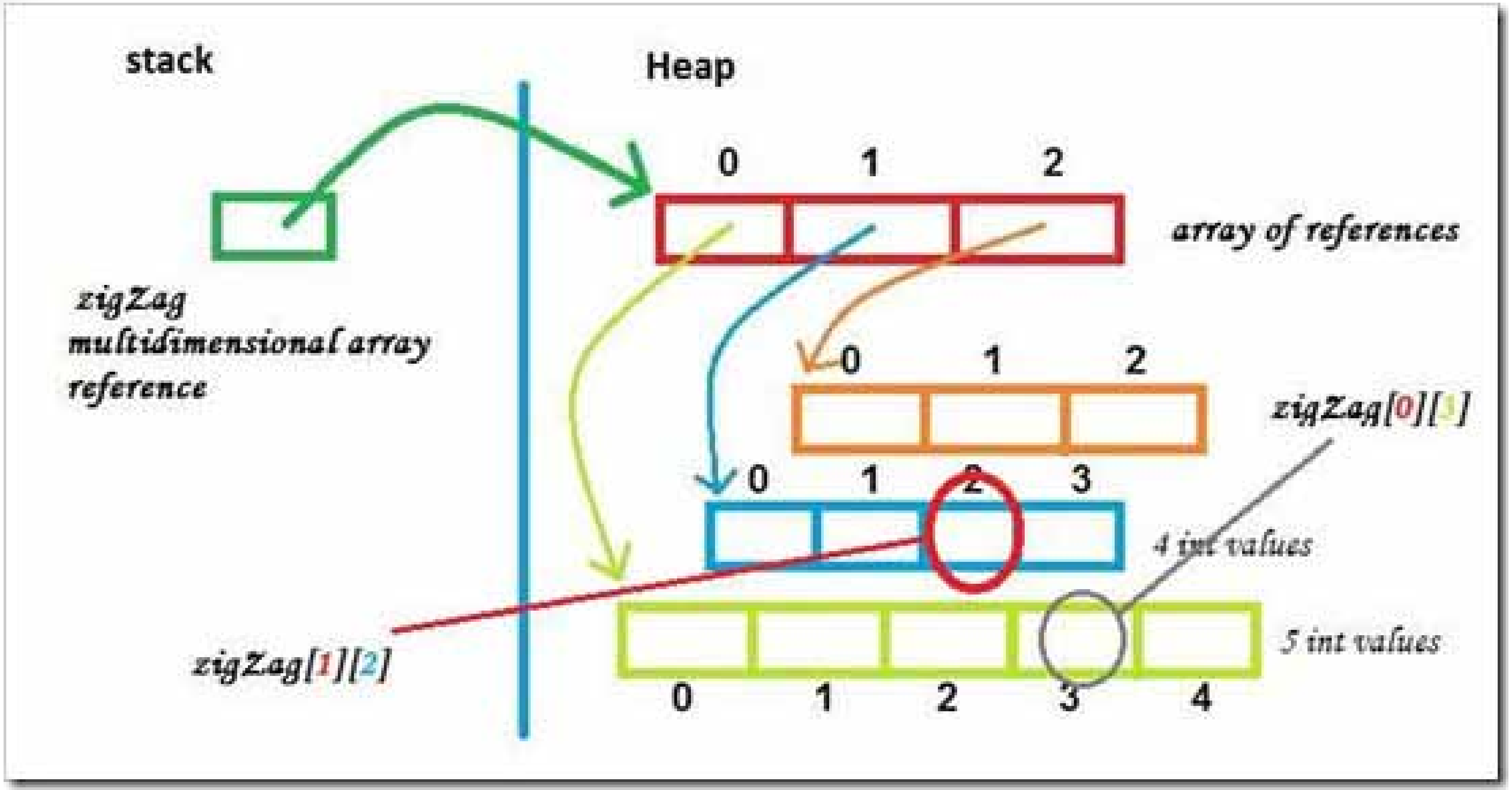
4 arrays,
each with 3
int values
number of elements for each second array
number of columns
number of elements for the first array
number of lines

© 2010 www.itcsolutions.eu

Java fundamentals
Two-dimensional Arrays



Java fundamentals
Two-dimensional Arrays



Java fundamentals
Enums

SINTAX: enum enum_name { constants list}

- a colection of constant values;
- is NOT a string or an int;
- enums are like classes; can contain constructors, methods, variables
- enum constants can send arguments to the enum constructor

Java fundamentals

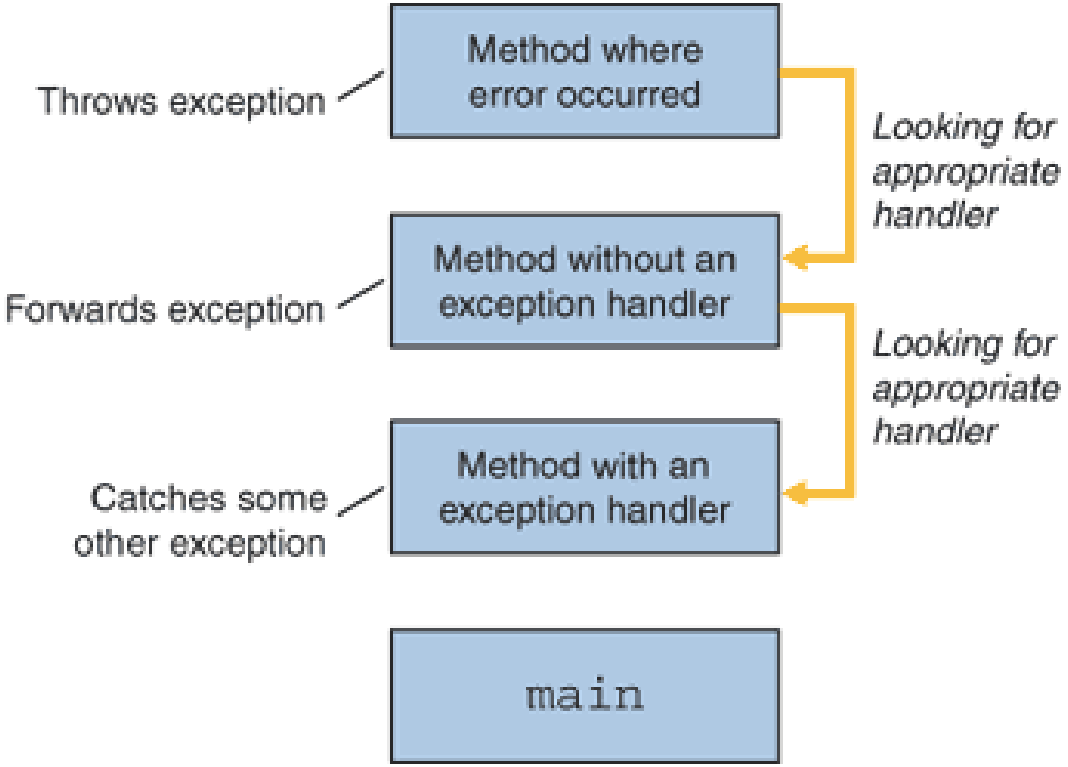
- try-catch-finally
- References
- Methods input parameters
- Classes
- Shallow copy vs Deep copy

Java fundamentals exceptions

- exception - the situation where processing certain input data is not managed or is not possible (eg dividing by 0, reading outside the bounders of an array)
- allows management of exceptional situations that lead to immediate termination of the program
- necessary to achieve robust and reliable program s
- implemented through try, catch, finally and throw
- allows the management of system exceptions

1. *checked exception* (exceptii verificate) = NU trec de compilare. Se poate prevede mecanism de “recovery”. Musai mecanism try-catch.
2. *errors* (erori) = trece de compilare DAR nu se poate prevede functionare defectuasa (e fizic stricat hard-diskul si la deschiderea de fisier se arunca ‘java.io.IOError’). De obicei nu exista mecanism de try-catch.
3. *runtime exception* (exceptii la rulare) = trec de compilare DAR din cauza logicii de la dezvoltare defectuase rezulta din calcule numitor=0 si mai departe o impartire la 0. Se poate utiliza try-catch, dar mai bine se elimina bug-ul de reuseste ca din calcule sa rezulte numitor=0.

- 2+3 = *unchecked exception*


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Java fundamentals exceptions

```
try
        {//statements}
catch(exception_type_1)
        { //particular statements}
catch(exception_type_2)
        {// particular statements}
catch(Exception)
        { //general statements}
finally
        {//must-do statements}
```

Java fundamentals exceptions

try block{…}

- contains the sequence of code that generate exceptions;
- has at least one catch block;
- between the try block and catch blocks there are no other instructions

catch block(exception_type exception)

- catch and manage an exception of *exception_type* type

exception_type is an instance of a class derived from Exception (InputMismatchException, ArrayIndexOutOfBoundsException)

Java fundamentals exceptions

catch(Exception err)

- catch and manage all exceptions

finally{…}

- contains code sequence that it is executed whether or not the try block has generated exceptions and whether they were or were not managed in other catch blocks;

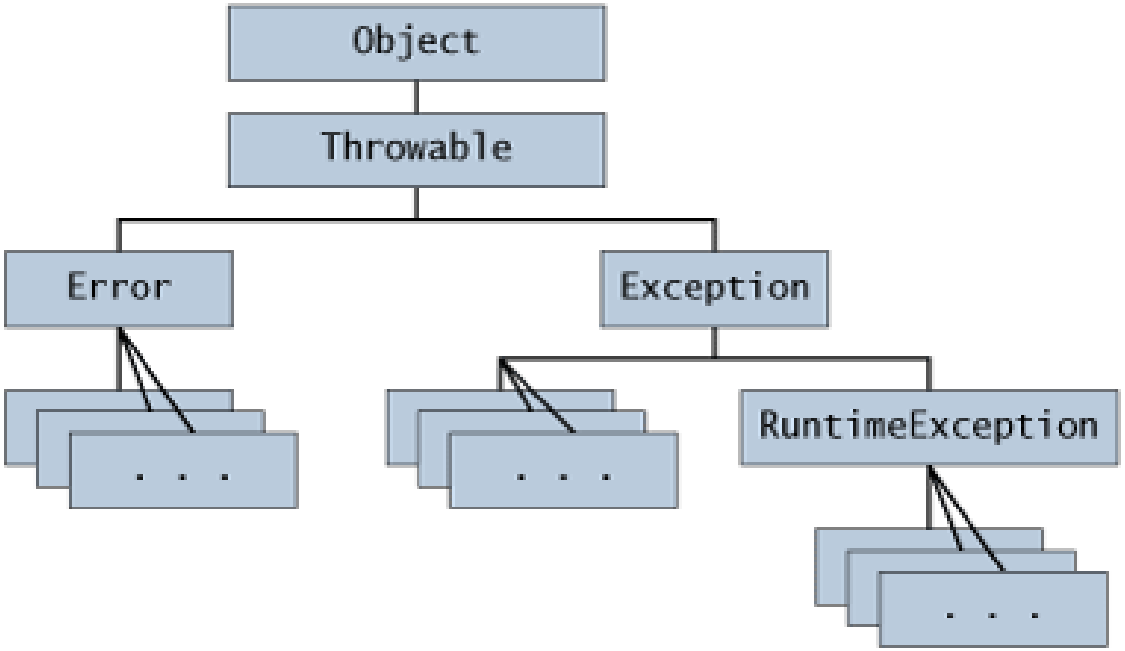
Java fundamentals exceptions

catch blocks are defined in the ascending order of the caught exceptions generality level

```
try { … }
catch(exception_type_1){…}
catch(exception_type_2){…}

…
catch(Exception err){…}
```

Java fundamentals exceptions

- Try-catch-finally blocks can be included in other try blocks;
- The programmer can define its own exceptions with classes derived from Exception;
- throw function generates a new exception
- local variables and try – “the try creep”


Object
Throwable
Error
Exception
RuntimeException
. . .
. . .
. . .

Java fundamentals exceptions

```
public class DividebyZeroException extends
   Exception
{
public DividebyZeroException() {   }
public DividebyZeroException(String msg) {
   super(msg);
   }
}
```

Java fundamentals methods input

- methods may receive input parameters that are primitive values or references
- by value -> copies the parameter value on the function stack
- by reference -> copies the address parameter on the function stack
- ATTENTION method arguments are always passed by their value; methods parameters are always copies of the arguments;

Java fundamentals
main arguments

SINTAX:

public static void main(String[ ] args)

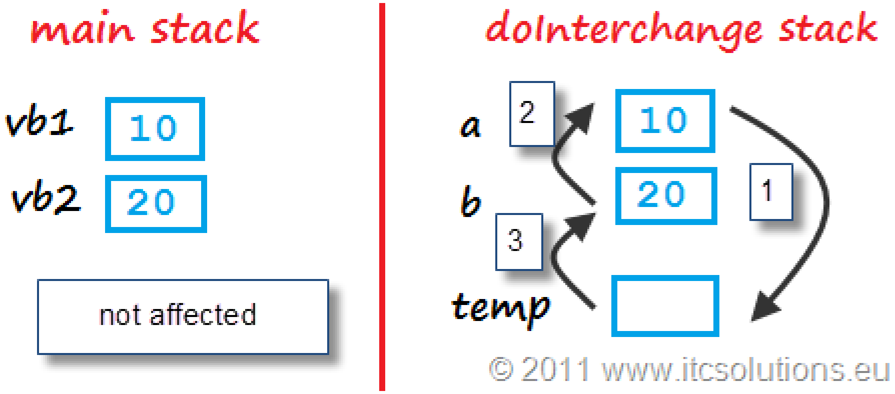
- it’s an array of Strings;
- the first value is not the executable name
- you must validate the number of inputs

java.exe hello1.class Popescu Gigel <-> args = {“Popescu”, ”Gigel” }

Java fundamentals
methods input

- transfer parameters by their value

```
public static void doInterchange(int p, int q){
        int t = p;
        p = q;
        q = t;
   }
```



Java fundamentals methods input

- transfer parameters by their reference

How to do that for a primitive variable ?

Remember. Method arguments are always passed by their value; methods parameters are always copies of the arguments.

Java fundamentals methods input

- transfer parameters by their reference

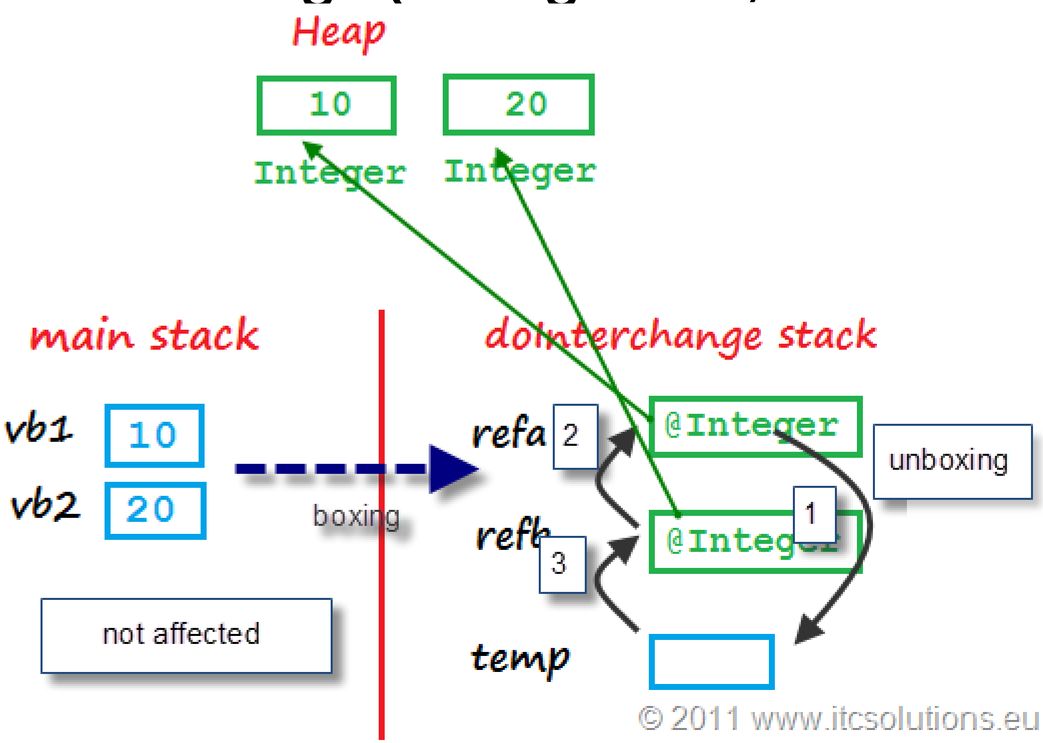
How to do that for a primitive variable ?

1. NO way to get the address of a primitive
2. NO way to use primitive wrappers – why ?
3. Use an array
4. Define your own wrapper class

Java fundamentals
methods input

- transfer parameters using wrappers

```
public static void doInterchange(Integer ra,
   Integer rb){
         int temp = ra;
         ra = rb;
         rb = temp;
   }
```



© 2011 www.itcsolutions.eu

Java fundamentals methods input

- methods can have variable length inputs

```
public static double sum(double... list)
  {
      double s = 0;
      if (list.length == 0) return s;
      for(int i = 0;i<list.length;i++)
      {
         s += list[i];
      }
      return s;
  }
```

Java fundamentals methods input

Shadowing a variable:

- shadowing occurs when you define a method with input parameters that have names identical to a static variables or instance variables

Java fundamentals - Packages

- packages allows different programmers to name their classes independently;
- the package statement must be the first line (before import statements) in the file
- if you want to use only one class from another package you define the import like this:

```
import packageName.className;
```

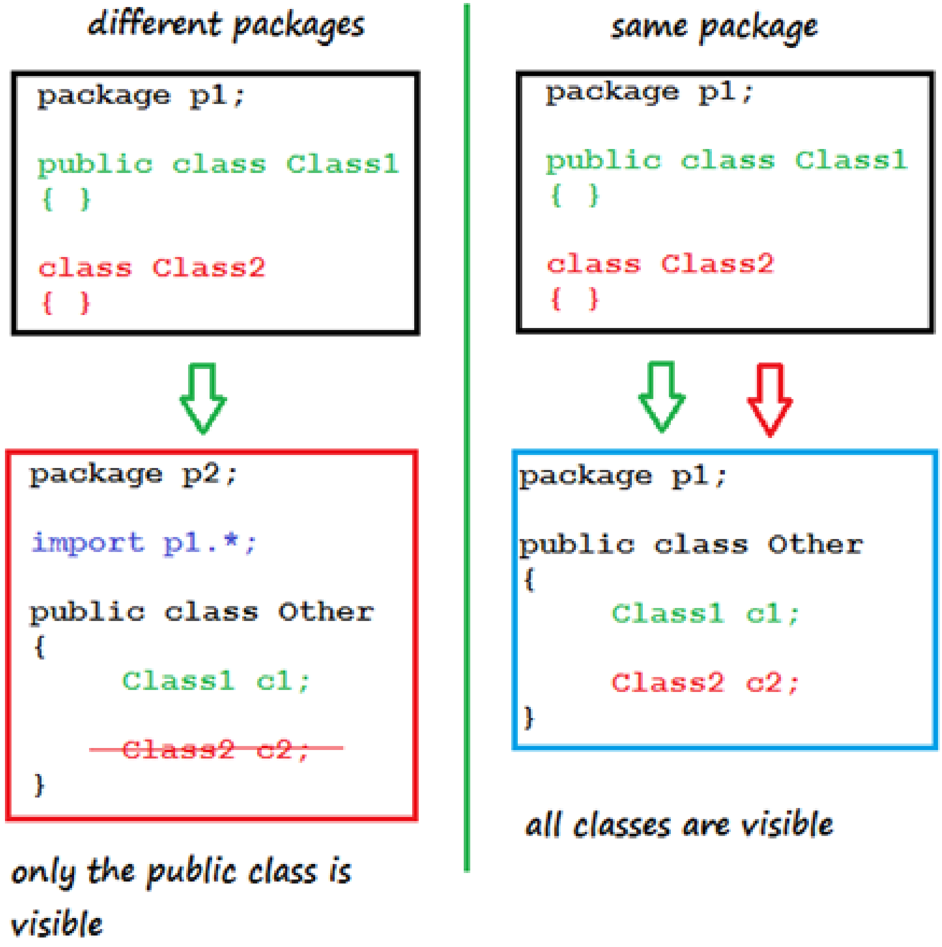
- if you want to use all classes from another package you define the import like this:

```
import packageName.*;
```

Java fundamentals - Packages

Package class modifiers are:

- **default** (when you don't use anything) - - the class is visible in the package;
- **public** - the class is visible anywhere



different packages

```
package p1;

public class Class1
{ }

class Class2
{ }
```

```
package p2;

import p1.*;

public class Other
{
      Class1 c1;

      ~~Class2 c2;~~
}
```

only the public class is visible

same package

```
package p1;

public class Class1
{ }

class Class2
{ }
```

```
package p1;

public class Other
{
      Class1 c1;

      Class2 c2;
}
```

all classes are visible

Java fundamentals
Classes

Object Oriented Programming Concepts:

- each object contains data (attributes / fields/instance variables) defined in class;
- the class defines a number of functions (methods / operations) that can be applied to objects, they define the object interface;
- data objects are hidden and can only be accessed by functions defined in class - encapsulation;
- objects are created by instantiating the class;
- abstraction (class definition) is to decide what attributes and methods are supported;
- object state is defined by its attributes;
- object behavior is defined by its methods;
- the term of passing a message to an object is equivalent to invoking the method;

Java fundamentals
Classes

Object Oriented Programming concepts in Java:

In Java all objects are managed only by references.

Java fundamentals
Classes

Sintax for defining a class:

```
[attributes][acces_modifier] class class_name
    [extends base_class][implements interface1,
    interface2, ...]
{
    access_modifier attribute1;  //instance variable
    access_modifier attribute2; //instance variable
    …
     access_modifier method1;
};
```

Java fundamentals
Classes

acces_modifier:

- public – accesibile
- private – not accesibile
- protected – accessible only in subclasses

attributes:

- final – the class can not be derived
- abstract – abstract class

Java fundamentals Classes

Instance members & methods visibility for access modifiers

Visibility	Public	Protected	Private	Default
Same class	X	X	X	X
Class in same package	X	X		X
Subclass in same package	X	X		X
Subclass in other package	X	X		
Class outside the package	X			

Java fundamentals
Classes

- extends – allows derivation from a base class (more in the tutorial about the derivation/inheritance)
- implements – allows the derivation from one or more interfaces (more in the tutorial about interfaces)

Java fundamentals
Classes

Regarding attributes we can define in a class:

- instance variables or attributes of objects;
- static variable – a type of “global variables”.

Regarding methods (functions) we can define in a class:

- constructor methods;
- access functions(get and set);
- processing methods;

Java fundamentals
Classes

- in a Java file, *.java* , can be defined several classes;
- in a Java file, can be defined only one public class;
- the Java source file containing the public class has the same name as the public class (case sensitive); *Book.java* contains public class *Book* ;
- the code block of a class is defined by { and };
- if in a source file, *.java,* are defined more classes, then by compiling the source file there are obtained bytecode files, *.class,* for each class.

Java fundamentals
Classes

How to construct objects:

- objects are constructed by the *new* operator that will call the class constructor (with or without parameters):

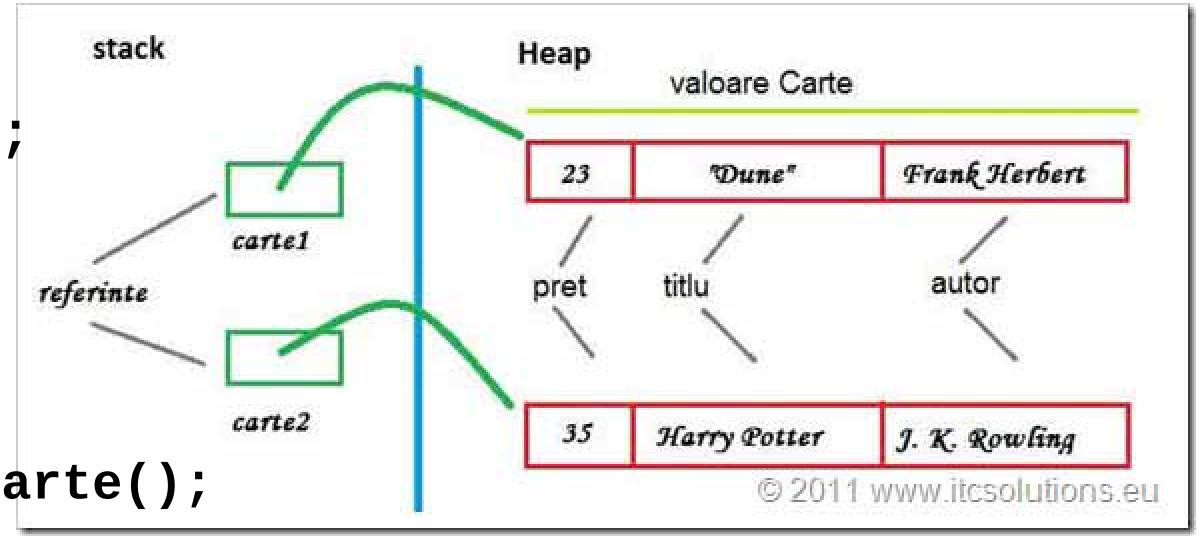
```
class_name reference = new class_name();
```

How to access object methods and attributes:

- the object has access to its attributes and methods (not static ones) through the . (point) operator.

Java fundamentals
Classes

```
class Book{
    //define attributes - instance variables
    float price;
    String title;
    String author;
}
…


Book book1 = new Carte();
book1.price = 23;
book1.title = "Dune";
book1.author = "Frank Herbert";
```



Java fundamentals
Classes

constant attributes:

- defined with final
- you can’t change their value once they have been initialized;

Sintax:

```
class Test
{
   public final int attribute_1 = 10;
//MUST be initialized in constructor
  public final int attribute_2;
}
```

Java fundamentals
Classes

constant attributes:

- are initialized in the constructor or (NOT and) at definition
- NOT allowed to change their value once they have been initialized;
- are equivalent to const variables in C++

Java fundamentals
Classes

static attributes:

- defined with static;
- defining attributes that does not belong to an object;
- can be final;
- initialization is done at definition or in a initialization block
- are considered variables defined at the class level, not attributes of an object;
- somehow global variables;
- are accessed by class name;

Java fundamentals
Classes

static attributes:

sintax:

```
class Test
{
    public static int attribute_1 = 10;
    public static final int attribute_2;
    //initialisation block
    //when is executed ?
    {
        attribute_2 = 45;
    }
}
```

Java fundamentals
Classes

this reference:

- address of the object that is calling a member method;
- all class members methods (NOT the static ones) receive this reference;

Java fundamentals
Classes

member functions:

- define the object interface;
- allow access to object attributes - encapsulation;
- define object behavior;
- special category of functions: constructor, copy constructor;
- Particular types: static;

Java fundamentals
Classes

static functions:

- define functions that do not belong to an object;
- are "global function", which belong to a class of objects;
- only have access to other static members of the class;
- are called by class specifier: **class_name.**
- they DO NOT receive *this* reference;

Java fundamentals
Classes

special functions:

- constructor
- copy constructor
- there are NO destructors – only finalize()
- access functions (get and set)
- can NOT override operators (like in C++)

Java fundamentals
Classes

Constructors:

- Main role: creates objects in Heap
- Secondary role: initialize object attributes (instance variables)
- Types:
  - – Default (without parameters)
  - – with input parameters

Java fundamentals
Classes

Constructors:

- have a name identical with the class name;
- DON'T have a explicitly return type because they return by default the address of newly created objects;
- defined in the public zone (NOT true for Singleton design pattern);
- only if there are no constructors the default form is generated by the compiler;

Java fundamentals
Classes

Constructors:

- syntax:

```
class class_Name {
public class_Name( ){…}
};
```

- use (because objects are managed by references, an object is created using *new* operator):

```
public static void main () {
class_Name object_1 = new class_Name();
class_Name object_2 = new class_Name(input params)
}
```

Java fundamentals
Classes

Destructor methods:

- in Java the are NOT destructors
- a function called at the destruction if the object is finalize() which is inherited from Object (it is not recommended to overload it)
- the object memory is released by Garbage Collector (JVM routine)

Java fundamentals
Garbage Collector

- controlled by JVM;
- collects not used objects:
  - nulling a reference;
  - reassigning a reference variable
  - isolating a reference
- can be called with System.gc() – not recommended

Java fundamentals
Classes

Copy Constructor:

- Main role: creates objects in Heap and initialize them with values from an existing object
- does NOT have an implicit form
- is used explicitelly
- low importance than the C++ version

Java fundamentals
Classes

Copy constructor:

- syntax:

```
  class class_name{
            public class_name(class_name existing_object){…}
                };
```

- use:

```
    public static void main (){
    class_name object_1 = new class_name(…);
    class_name object_2 = new class_name(object_1);
    }
```

Java fundamentals Classes

In Java you can’t overload operators

Java fundamentals
Classes

= operator

- copies bit by bit source value into destination memory area (the two areas are identical in structure and type);
- for objects, it copies the source reference value over the destination object reference

Java fundamentals
Classes

properties methods: getters and setters

- allow access (read / write) to class private attributes
- involve validating input
- are defined in the public area
- defined by two methods
- read method has a get prefix (standard na
- write method has a set prefix;

Java fundamentals
Classes

```
class Test{
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Java fundamentals

Shallow copy vs. Deep copy

Shallow copy

- copies reference values between them;
- done implicitly with =

Deep copy

- copies objects values (not their references)
- done by special methods (copy-constructor, clone, etc)

Java fundamentals

Shallow copy vs. Deep copy

How to implement *clone* method:

- overrides *Object* inherited method:
- write your own method;

```
@Override
public Test clone()
{
    …
}
```

Java fundamentals
Shallow copy vs. Deep copy

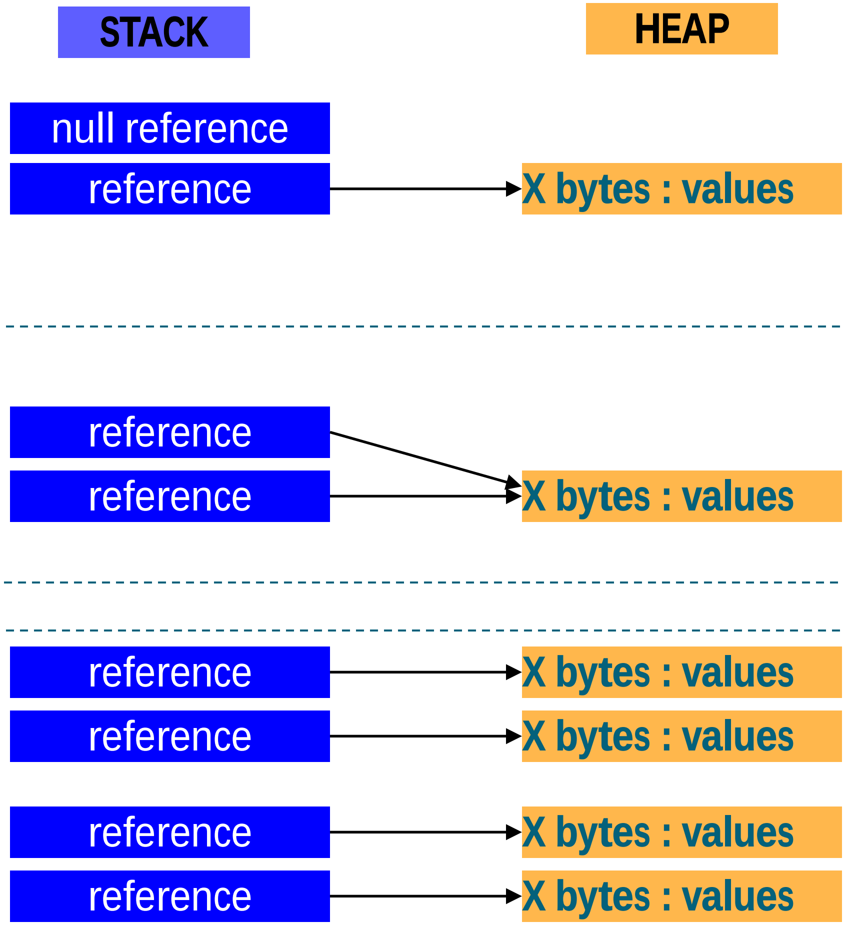
```
Auto a1;
Auto a2 = new Auto();
```

Shallow copy

```
a1 = a2
```

```
a1 = new Auto(a2)

a1 = (Auto)a2.clone()
```

Deep copy



Java fundamentals

- Overriding
- Inheritance
- Interfaces / abstract classes
- Virtualization
- Callback and Events

Java fundamentals
Inheritance

it is implemented when there is a ***is a*** relation between the subclass and the base class;

you can inheritance only one base class:

```
class SpecialProduct extends Product
{         private float _discount;
                         …
}
```

- calling the base class constructor is made through *super*:

```
public SpecialProduct(double cost, float profit, float discount)
          {
                    super(cost,profit);
                    if(discount>0) _discount=discount;
          }
```

Java fundamentals
Inheritance

In Java all classes inheritance Object

Java fundamentals
Inheritance

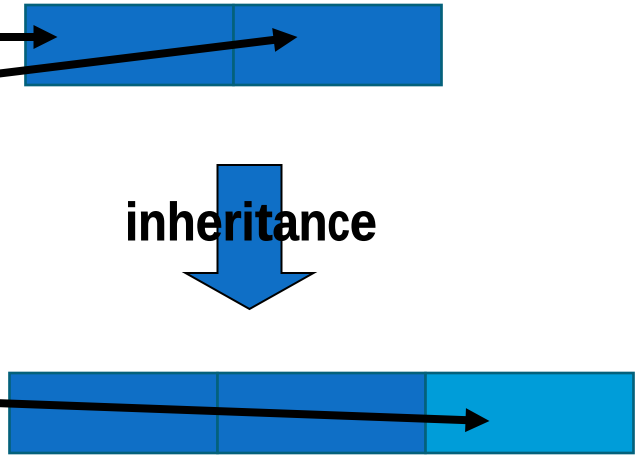
Methods of Object:

- boolean equals (Object obj)
- void finalize()
- int hashCode()
- final void notify()
- final void notifyAll()
- final void wait()
- String toString()

Java fundamentals
Inheritance

through inheritance the subclass gets all the methods and attributes

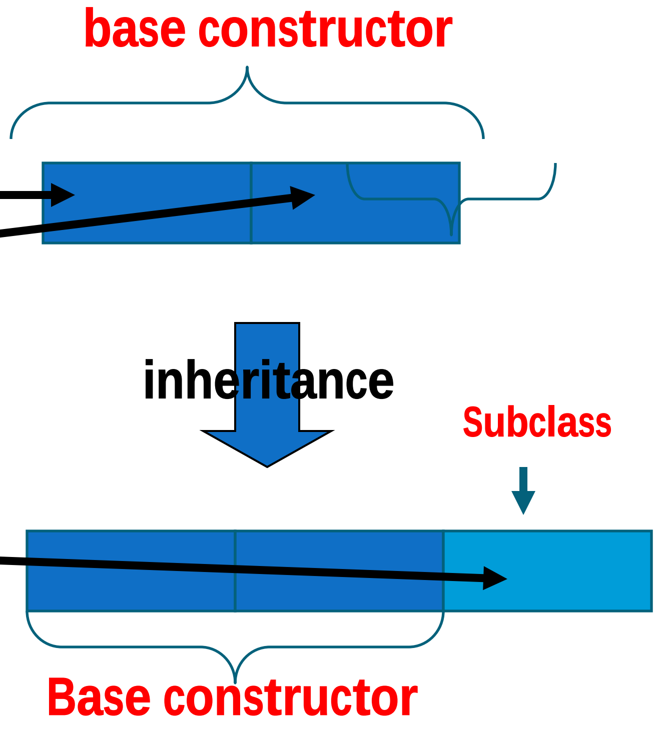
```
class Base{
        int attribute1;
        int attribute2;
};
class Subclass: Base{
        int new_attribute;
};
```



Java fundamentals
Inheritance

each constructor manages the area of its class

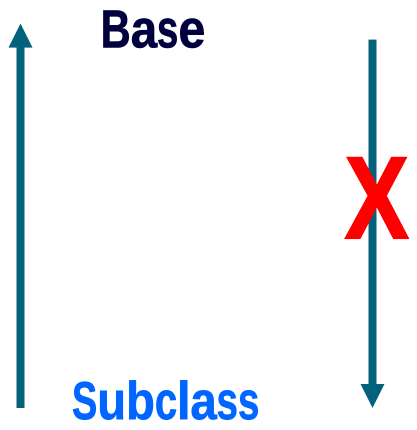
```
class Base{
        int attribute1;
        int attribute2;
};
class Subclass: Base{
         int new_attribute;
};
```



Java fundamentals
Inheritance

UPCASTING – it is allowed to manage a subclass reference using a base reference

```
class Base{
        …
};
class Subclass : Base{
        …
};
```

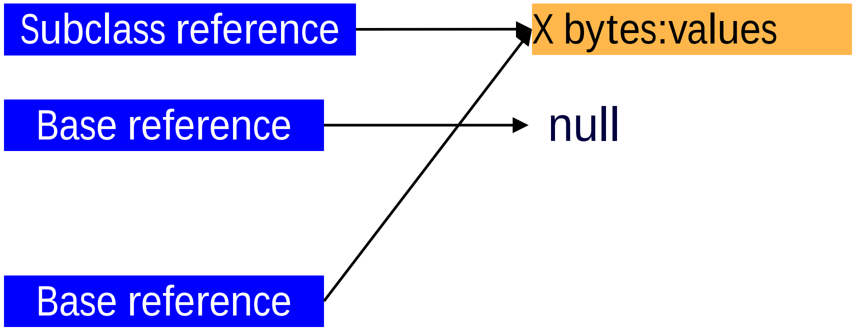


Java fundamentals

Inheritance

UPCASTING

```
Subclass sc = new Subclass();
Baza bc;

Baza bc = sc;
```



Subclass reference

X bytes:values

Base reference

null

Base reference

Java fundamentals
Inheritance

in the base class and the subclass you can define methods with the same header - overloading

```
class Base{
        int Method1(int a){…}
};
class Subclass: Base{
        int attribute_new;
        int Method1(int a){…}
        int Method2(int a){ super.Method1(a);}
};
```

call to the base class

Java fundamentals
Inheritance

In Java all methods are virtual

Java fundamentals
Inheritance

UPCASTING & overriding

```
Subclass s1 = new Subclass()
Base b1 = s1;

b1.Methode1(5);
s1.Methode1(5);
```

Subclass method;

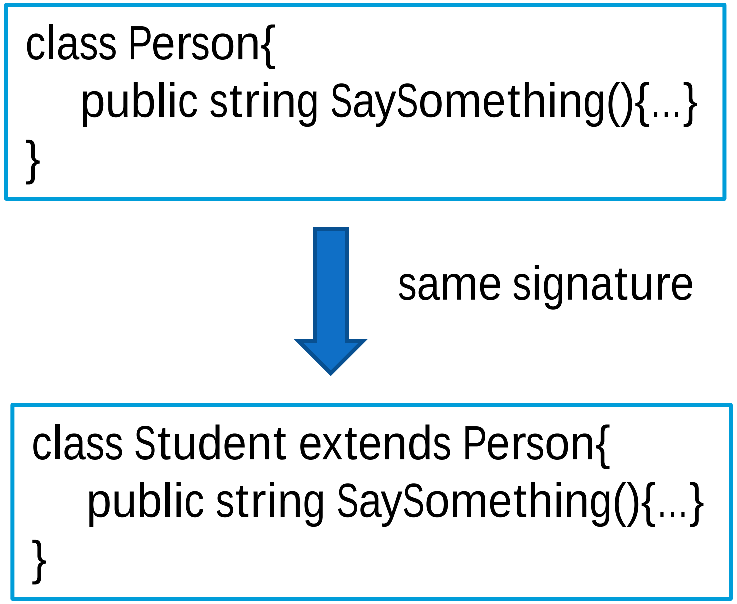
Java fundamentals
Inheritance

POLIMORPHYSM (**same thing, many functionalities**) :

- OVERLOADING methods in a class
- OVERRIDE virtual methods in subclasses

Java fundamentals
Inheritance

Overriding

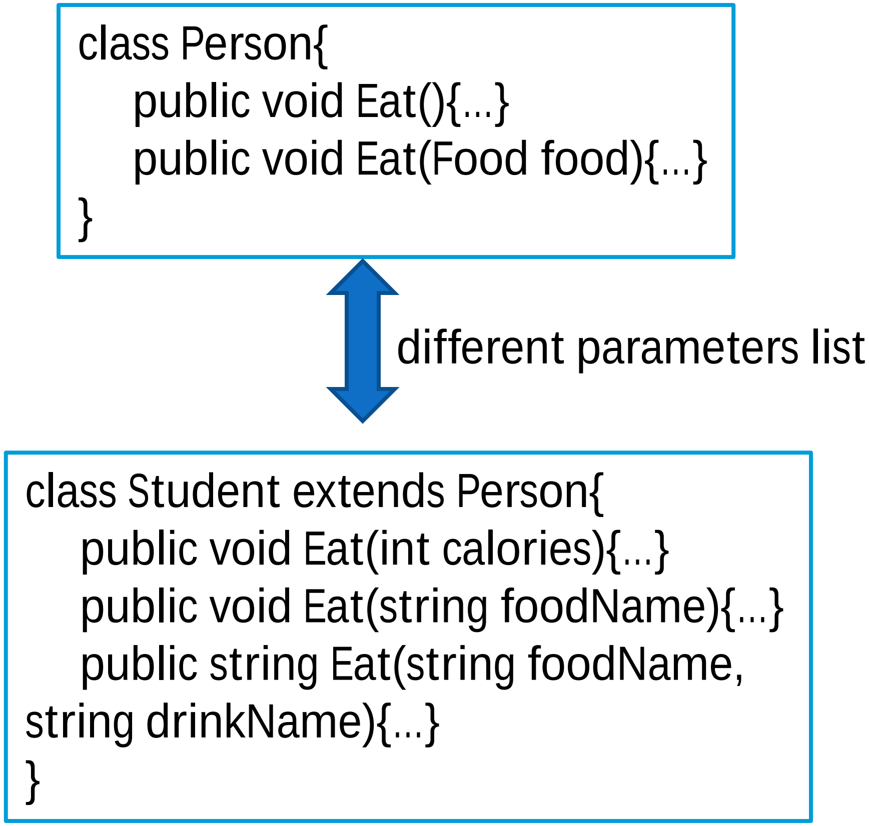


```
class Person{
    public string SaySomething(){…}
}
```

same signature

```
class Student extends Person{
    public string SaySomething(){…}
}
```

Overloading



```
class Person{
    public void Eat(){…}
    public void Eat(Food food){…}
}
```

different parameters list

```
class Student extends Person{
    public void Eat(int calories){…}
    public void Eat(string foodName){…}
    public string Eat(string foodName,
string drinkName){…}
}
```

Java fundamentals

Inheritance

Overriding Object methods ≠ overloading them

Java fundamentals
Inheritance

Inheritance vs Coposition

```
class Vehicle{
        …
};

class Auto extends Vehicle{
        …
};
```

the subclass *is a special form of the base class*;

Java fundamentals
Inheritance

Inheritance vs **Coposition**

the class *has* an instance variable of other class type ;

```
class Engine{
        …
};

class Auto {
        Engine engine;
};
```

Java fundamentals
Inheritance

- operator *instanceof* is used to test whether an object is an instance of some class;
- conversion from a base class reference to a subclass one is done with the cast operator:

```
if(b instanceof SpecialTest)
        sc = (SpecialTest)b;
```

Java fundamentals

Inheritance

PURE VIRTUAL methods - ABSTRACT:

- virtual functions that don’t have a body in the parent class
- defined with

abstract acces_type return type method_name(parameters);

- the subclass must override them, if it not abstract;
- parent class must be abstract;

Java fundamentals

Inheritance

PURE VIRTUAL methods - ABSTRACT:

```
abstract class Superclass{
public abstract int Method1(int a) ;
};

class Subclass extends Superclass{
public int Method1(int a){...}
};
```

Java fundamentals
Inheritance

Abstract classes:

- classes that contain at least one pure virtual function (abstract) – not required;
- can contain attributes and methods;
- an interface for classes that should define a set of common methods
- a contract between the owners of several classes that impose them to define common methods;
- contract is concluded by inheriting an abstract class;

Java fundamentals
Inheritance

Abstract classes:

- you can’t instantiate an abstract class;
- used for class hierarchies

```
abstract class AbstractClass{
        int attribute1;
        public abstract int Method1(int a);
};
....
AbstractClass ba1;
```

~~AbstractClass ba1 = new AbstractClass();~~

```
}
```

Java fundamentals
Inheritance

Final classes:

- you can’t inherit a final class;

```
final class Base {
        int attribute1;
        ...
};
class Subclass extends Base { }
```

compiler error

Java fundamentals
Inheritance

Interfaces:

- abstract classes that contain only abstract functions;
- an interface for classes that should define a set of common methods
- a contract between the owners of several classes that are necessary to define common methods series;
- contract is concluded by deriving from the interface using implements;
- are defined by the interface keyword;

Java fundamentals
Inheritance

Interfaces:

```
interface IOperations {
        void Operation1( );
        void Operation2( );
};
class Base implements IOperations {
        public void Operation1() {…}
        public void Operation2() {…}
 }
```

override interface method

Java fundamentals

Abstract classes

- contain abstract methods + attributes + non-abstract methods
- a class can extend only one base class (abstract or not)
- can be used as *reference type*

vs.

Interfaces

¢ contain only abstract methods

¢ a class can implement on or many interfaces

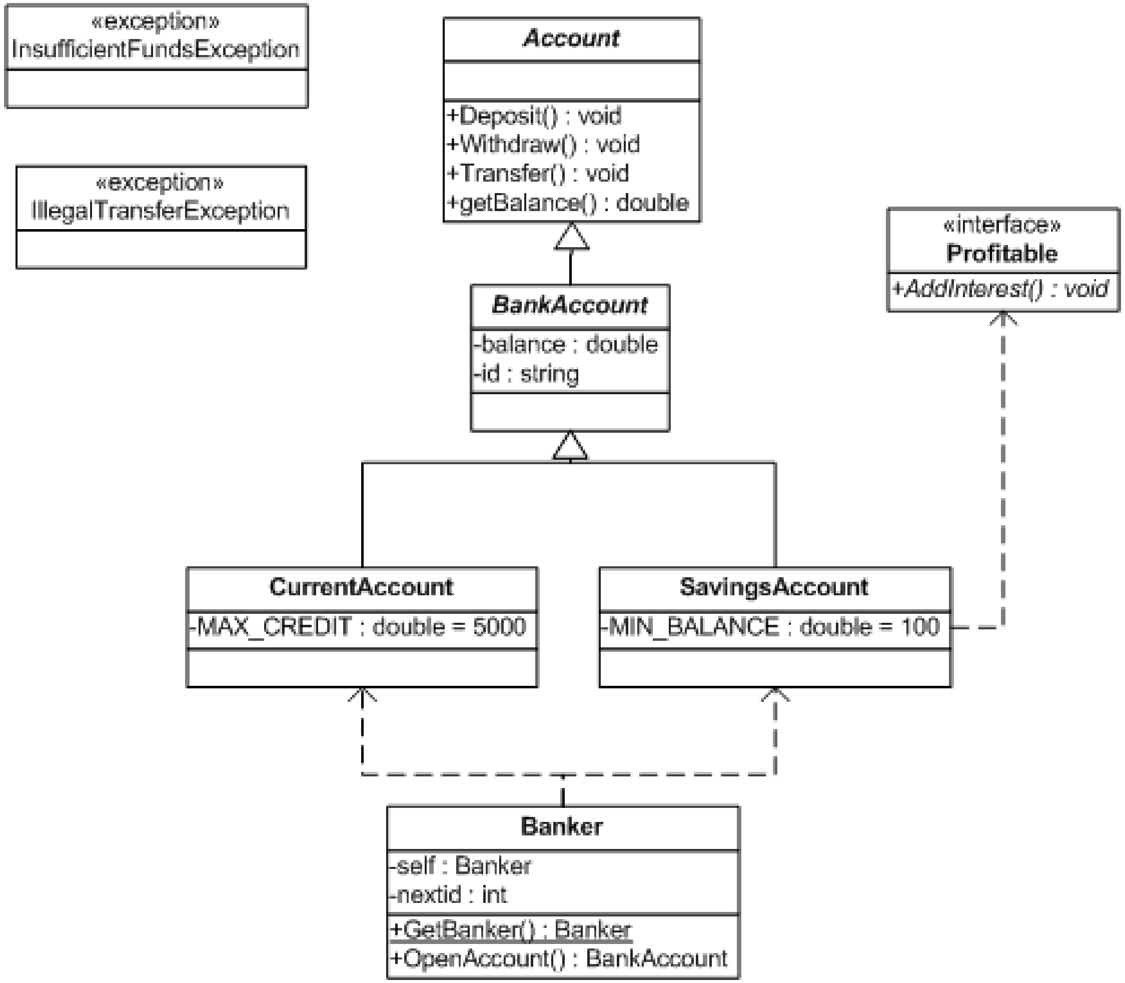
¢ can be used as *reference type*

Java fundamentals
Design patterns

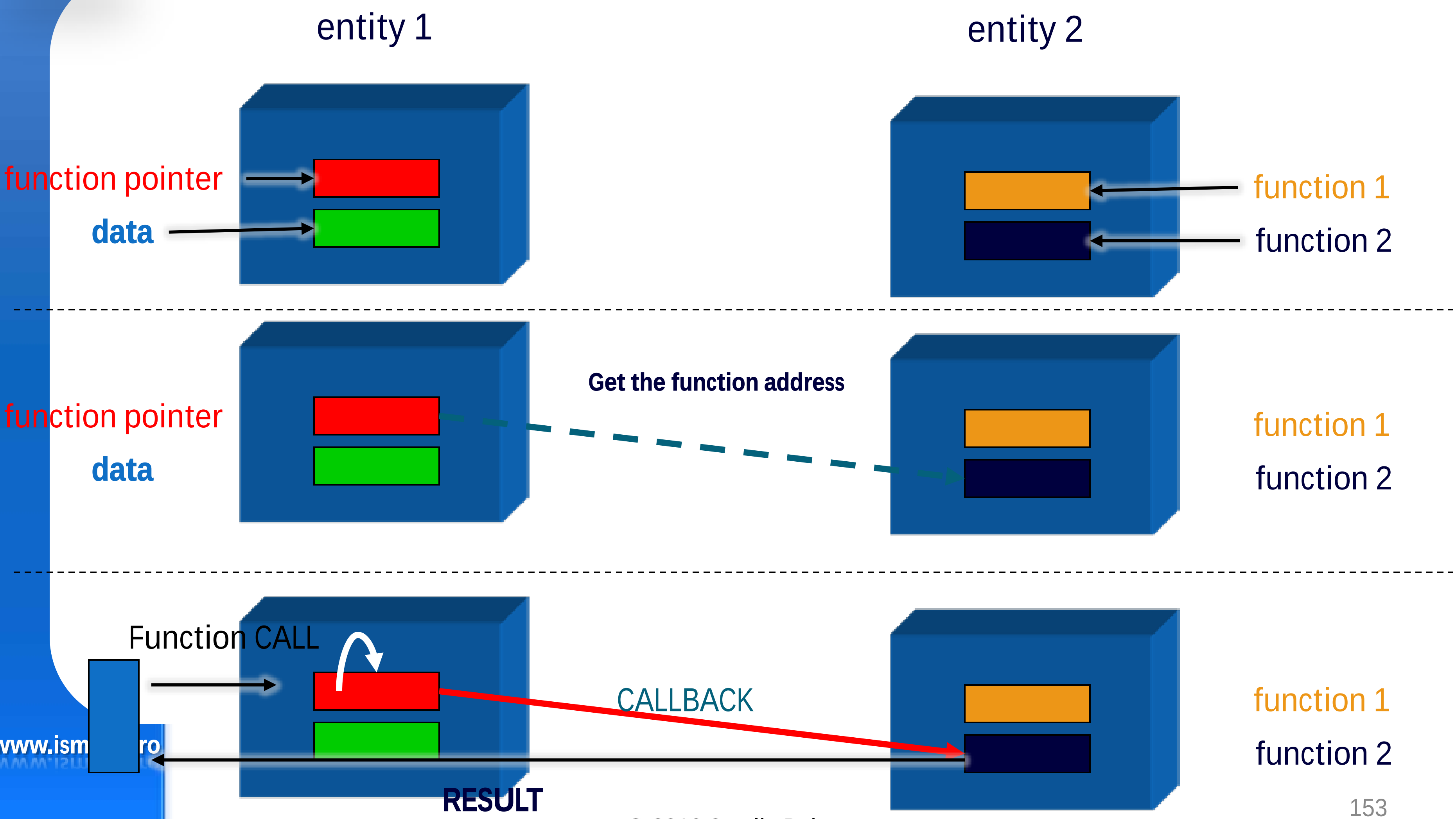
- Singleton – “Ensure a class only has one instance, and provide a global point of access to it”.
- Factory – “Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses”.

[Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - **Design Patterns Elements of Reusable Object-Oriented Software,** Addison-Wesley Pub Co; 1st edition (January 15, 1995)]

Java fundamentals
Design patterns

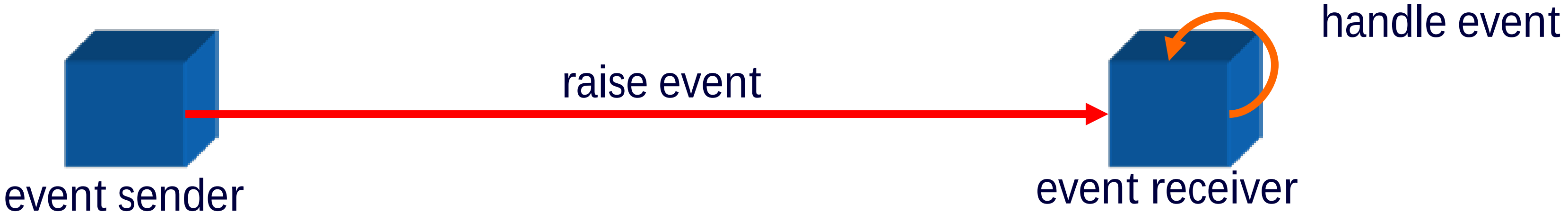


Java fundamentals
Callback



Java fundamentals
Callback & Events

- An event is a message sent by an object to announce something (GUI user interaction - mouse click, button click or program logic – application routines)



- In a event handle model, the object that manages the event (and raises it) doesn't know what method will receive and handle it; that's why it is needed something that will connect the source and the destination (in Java this is ????)

Java fundamentals
Callback & Events

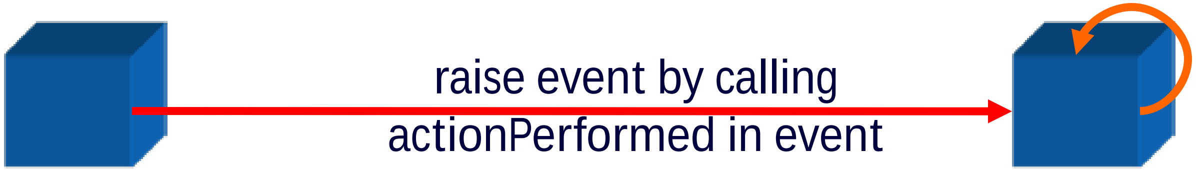
- There are NO pointers to functions (like in C/C++)
- There are NO new data types (like delegate and event in .NET)
- You have only INTERFACES

Java fundamentals
Callback & Events

- the Java standard approach for events and events handlers (for GUI) implies

```
public interface ActionListener extends
java.util.EventListener {
void actionPerformed( ActionEvent e );
}
```

has methods used to add/remove event receivers



raise event by calling actionPerformed in event receivers

handle event by executing **actionPerformed**

event sender

```
void addActionListener(ActionListener listener) { … }
void removeActionListener(ActionListener listener) { … }
```

event receiver implements **ActionListener**

Java fundamentals

- Collections and Generics
- Collection Interfaces
- Iterators
- Annotations

Java advanced - Generics

- Generics enforce compile-time safety on Collections or other classes and methods declared using generic type parameters
- uses <type name> syntax;
- between compiler-error and run-time error, the first is preferred;
- used mostly with generic collections

Java advanced - Generics

```
public class Box {

Object value; //generic reference

public void setValue(Object value)    {this.value = value;}
public Object getValue()    {return value; }
}
```

A generic class – classic approach

VS.

```
public class GenericBox<T> {
T value; //generic type
public void setValue(T value) { this.value = value;}
public T getValue()    {return value;}
}
```

A generic class – generics aproach

Java advanced - Generics

- Wildcard syntax allows a generic method to accept subtypes (or supertypes) of the declared type;
- The ? wildcard is called the *unbounded wildcard* and denotes that any type instantiation is acceptable;

```
List<?> anyList = new ArrayList<Date>( );
anyList = new ArrayList<String>( );
```

Java advanced - Generics

- A bounded wildcard is a wildcard that uses the **extends** keyword to limit the range of assignable types

```
List<? extends Date> dateList = new ArrayList<Date>( );
dateList = new ArrayList<MyDate>( );
```

Java advanced - Generics

Generic classes:

- Represent template classes (like in C++) – descriptions of classes with parameters;
- Can be adapted to real types (Java types + user defined);
- Creating instances, the JVM generates real classes;
- A generic class requires one or more type parameters

Java advanced - Generics

```
public class TestGenerics<T> {
        T instanceVariable;
        T[] array
        TestGenerics(T input){
           instanceVariable = input;
        }
        T getInstance(){
           return instanceVariable;
}

TestGenerics<int> ref = new TestGenerics<int>();
```

type variable

Java advanced - Generics

Generic methods:

- have a parameter type declaration using the <> syntax
- permit cresterea gradului de generalizare prin definirea de sabloane de functii
- syntax appears before the return type of the method:

```
      <T> return_type method(parameters)
ex: <T> T doSomething(int val1, T val2){}
```

Java advanced - Generics

Generic methods:

- unlike a generic class, it does not have to be instantiated with a specific parameter type for T before it is used
- initializare & utilizare:

```
int val1 = 10;
int val2 = 20;
int result doSomething(val1, val2);
```

Lab 9 – Generics

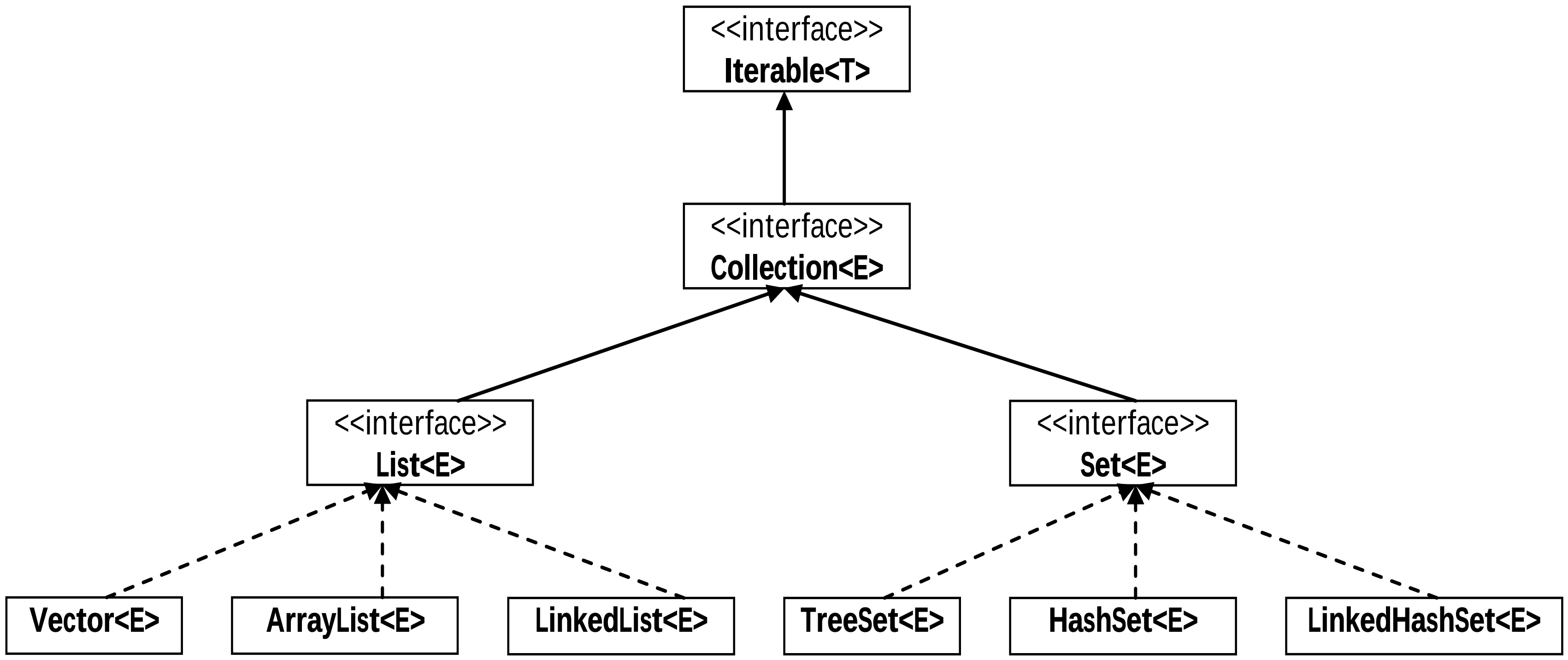
- **Work in: \Java\Labs\Advanced\**
- **Use: Eclipse**
- **Project: GenericBox**
- **Objective: Understand generics**
- **Problem: write and run a Java application that implement a generic container**

Java fundamentals
Collections

- to use own objects in collections you need to override Object methods:
  - – boolean equals (Object obj)
  - – int hashCode()
- to sort object you need to implement *Comparable* (int compareTo(Object)) or *Comparator*(int compare(Object one, Object two)) interface

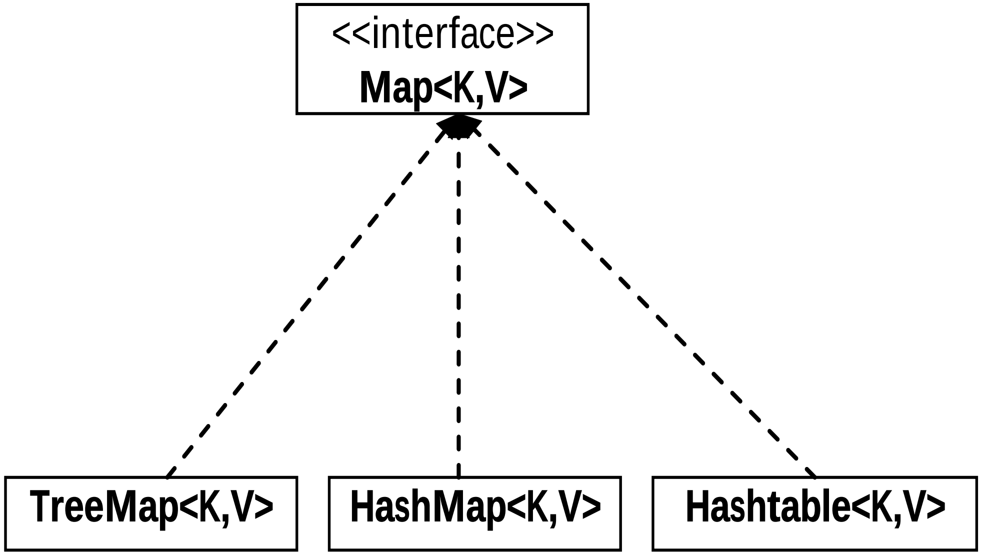
Java fundamentals

Collections



Java fundamentals

Collections



<<interface>>
Map<K,V>

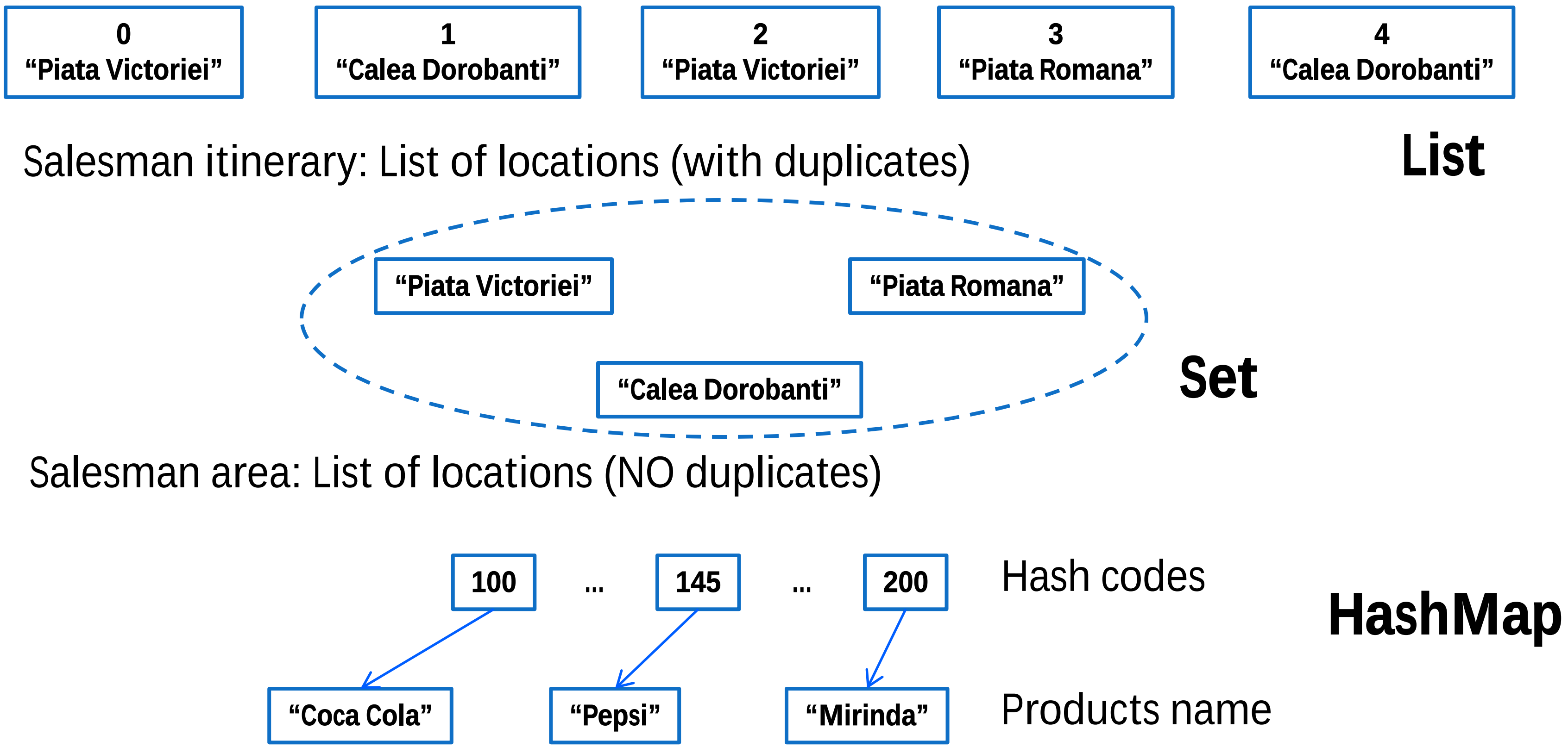
TreeMap<K,V>

HashMap<K,V>

Hashtable<K,V>

Java fundamentals

Collections



0 "Piata Victoriei"   1 "Calea Dorobanti"   2 "Piata Victoriei"   3 "Piata Romana"   4 "Calea Dorobanti"

Salesman itinerary: List of locations (with duplicates)

List

"Piata Victoriei"   "Piata Romana"   "Calea Dorobanti"

Set

Salesman area: List of locations (NO duplicates)

100 … 145 … 200   Hash codes

"Coca Cola"   "Pepsi"   "Mirinda"   Products name

HashMap

Salesman products managed by their name

Java fundamentals
Collections

- Lists – list of values (ArrayList, Vector, LinkedList)
- Sets – list of unique values (HashSet, TreeSet, LinkedHashSet)
- Maps – list of values with an unique ID (Hashtable, HashMap, TreeMap, LinkedHashMap)
- Queues – list o values processed in a specific order

Java fundamentals
Collections

- Types of collections:
- Sorted
- Unsorted
- Ordered
- Unordered - Hashtable

Java fundamentals

Collections

Class	Map	Set	List	Ordered	Sorted
HashMap	X				
Hashtable	X				
TreeMap	X			sorted	X
LinkedHashMap	X			by insertion	
HashSet		X			
TreeSet		X		sorted	X
LinkedHashSet		X		by insertion	
ArrayList			X	by index	
Vector			X	by index	
LinkedList			X	by index	
PriorityQueue				sorted	X

Java fundamentals
Collections

Iterator:

- objects used to manage the current position inside a collection
- Iterator
- ListIterator
- to define an iterator over your collection you must implement *Iterable* and *Iterator* interfaces

Java fundamentals
Collections

Algorithms:

- *polymorphic algorithms* implement different functionality provided by the Java platform;
- static methods in Collections class;
- most of the time used with List instances

Java fundamentals

Collections

Algorithms:

Algorithm	Method	Description
Sorting	sort()	Sorts a List by natural order or by a Comparator
Shuffling	shuffle()	Shuffles a Collection
Searching	binarySearch()	Searches a sorted list for a given value

Composition	frequency()
	disjoint()
		The frequency of a given value
		Number of common elements in 2 collections

Find extreme values	min()
	max()	

Java fundamentals
Collections

Algorithms:

Algorithm	Method	Description
Routine Data Manipulation	reverse()	Reverses the order of the elements in a List
	fill()	Overwrites every element in a List with the specified value
	copy()	Copies a source List into a destination one
	swap()	Swaps the elements at the specified positions in a List
	addAll()	Adds elements to a Collection

Java fundamentals
Annotations

- metadata for Java classes, methods, and fields;
- Used by compilers and even at the runtime
- Standard annotations:
  - @Deprecated
  - @Override
  - @SuppressWarnings(value="type")
- You can define your own annotations

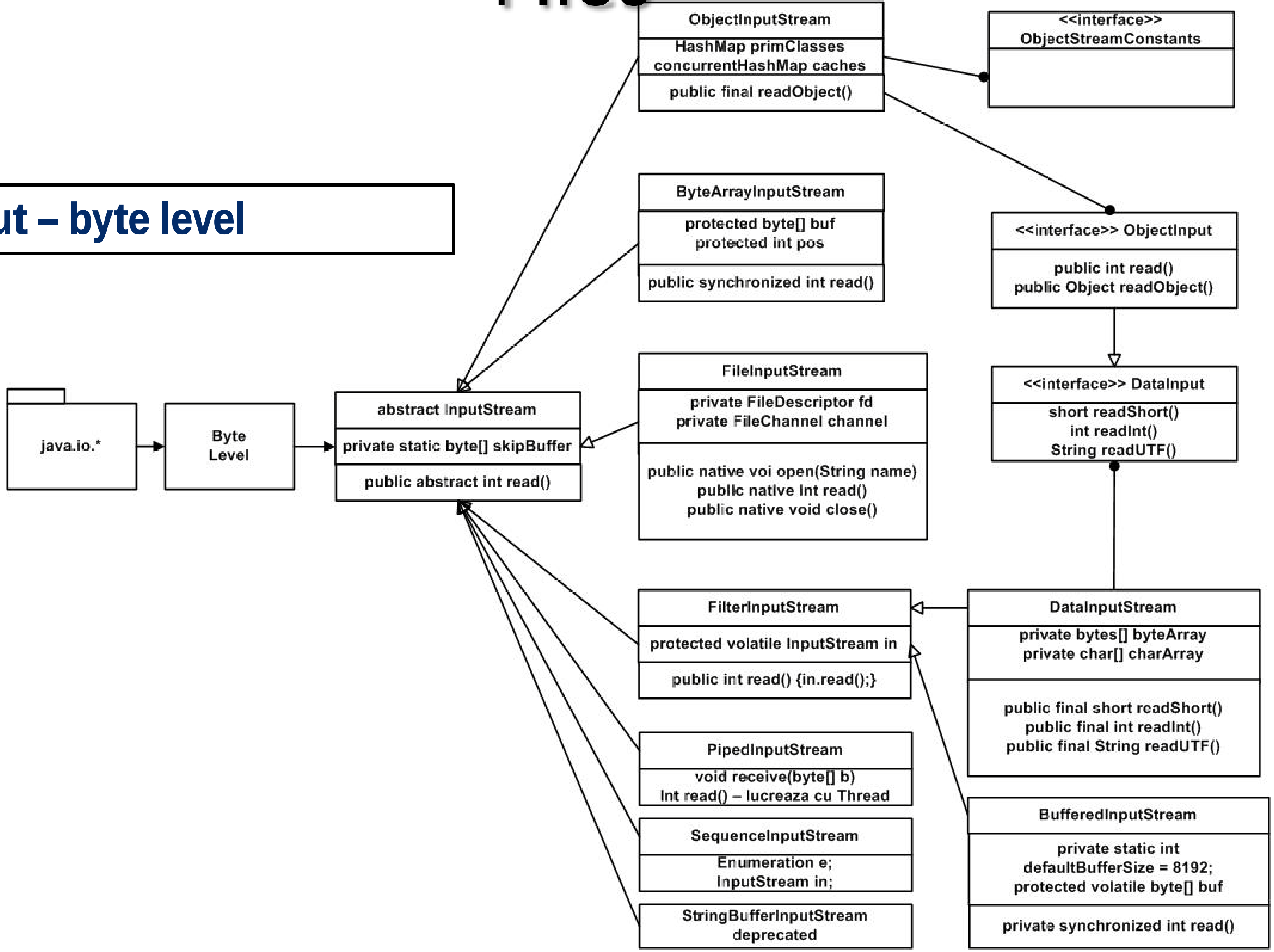
Java fundamentals
Files

- I/O operations are based on streams:
  - At byte level: InputStream, OutputStream;
  - at char level (in Java a char = 2 bytes): Reader, Writer
- files are managed by ***File*** objects

Java fundamentals

Files

Input – byte level



java.io.*

Byte Level

abstract InputStream
private static byte[] skipBuffer
public abstract int read()

ObjectInputStream
HashMap primClasses
concurrentHashMap caches
public final readObject()

<<interface>> ObjectStreamConstants

ByteArrayInputStream
protected byte[] buf
protected int pos
public synchronized int read()

<<interface>> ObjectInput
public int read()
public Object readObject()

FileInputStream
private FileDescriptor fd
private FileChannel channel
public native voi open(String name)
public native int read()
public native void close()

<<interface>> DataInput
short readShort()
int readInt()
String readUTF()

FilterInputStream
protected volatile InputStream in
public int read() {in.read();}

DataInputStream
private bytes[] byteArray
private char[] charArray
public final short readShort()
public final int readInt()
public final String readUTF()

PipedInputStream
void receive(byte[] b)
Int read() – lucreaza cu Thread

BufferedInputStream
private static int defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int read()

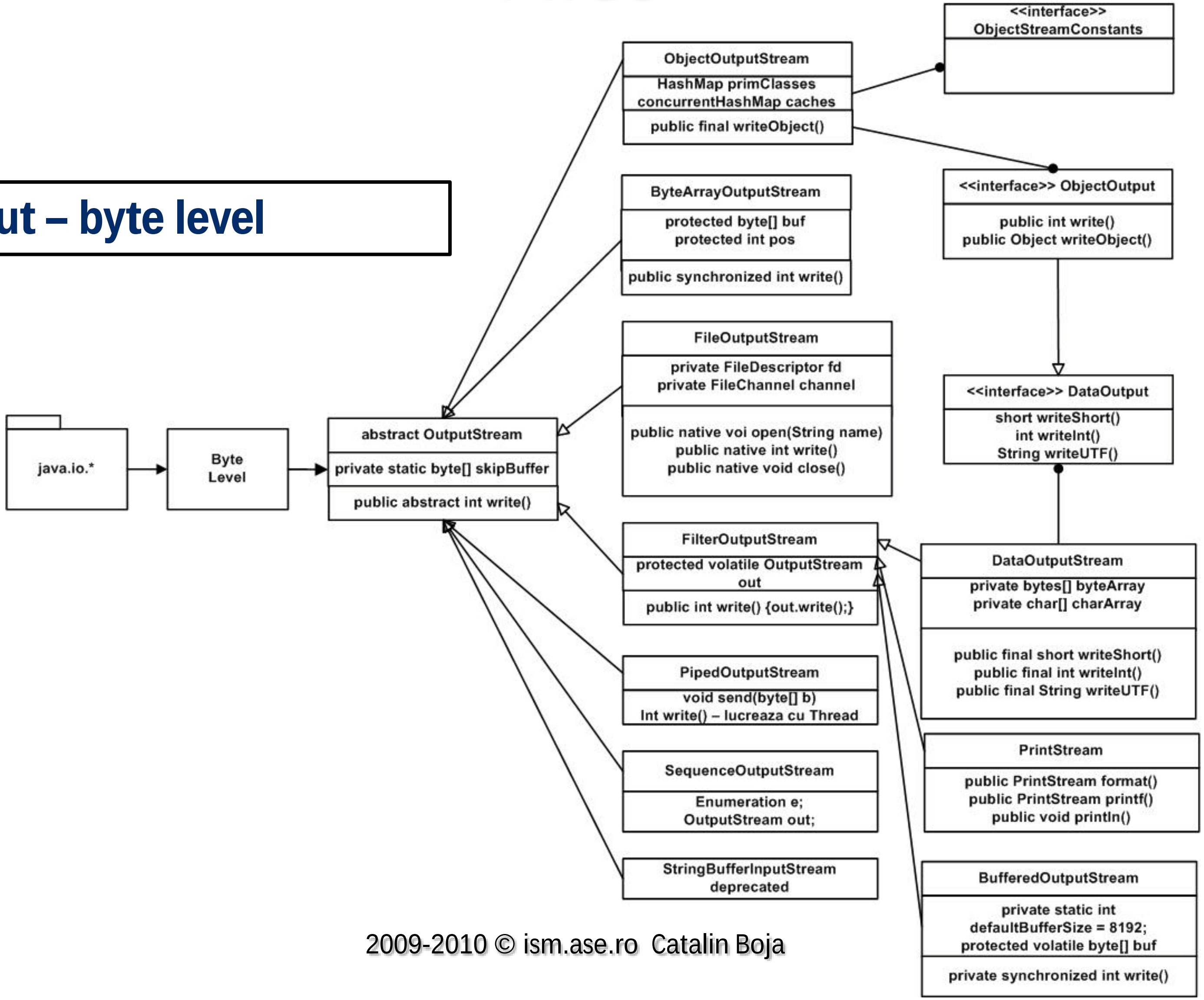
SequenceInputStream
Enumeration e;
InputStream in;

StringBufferInputStream
deprecated

Java fundamentals

Files

Output – byte level



java.io.*

Byte Level

abstract OutputStream
private static byte[] skipBuffer
public abstract int write()

ObjectOutputStream
HashMap primClasses
concurrentHashMap caches
public final writeObject()

<<interface>> ObjectStreamConstants

ByteArrayOutputStream
protected byte[] buf
protected int pos
public synchronized int write()

<<interface>> ObjectOutput
public int write()
public Object writeObject()

FileOutputStream
private FileDescriptor fd
private FileChannel channel
public native voi open(String name)
public native int write()
public native void close()

<<interface>> DataOutput
short writeShort()
int writeInt()
String writeUTF()

FilterOutputStream
protected volatile OutputStream out
public int write() {out.write();}

DataOutputStream
private bytes[] byteArray
private char[] charArray
public final short writeShort()
public final int writeInt()
public final String writeUTF()

PipedOutputStream
void send(byte[] b)
Int write() – lucreaza cu Thread

PrintStream
public PrintStream format()
public PrintStream printf()
public void println()

SequenceOutputStream
Enumeration e;
OutputStream out;

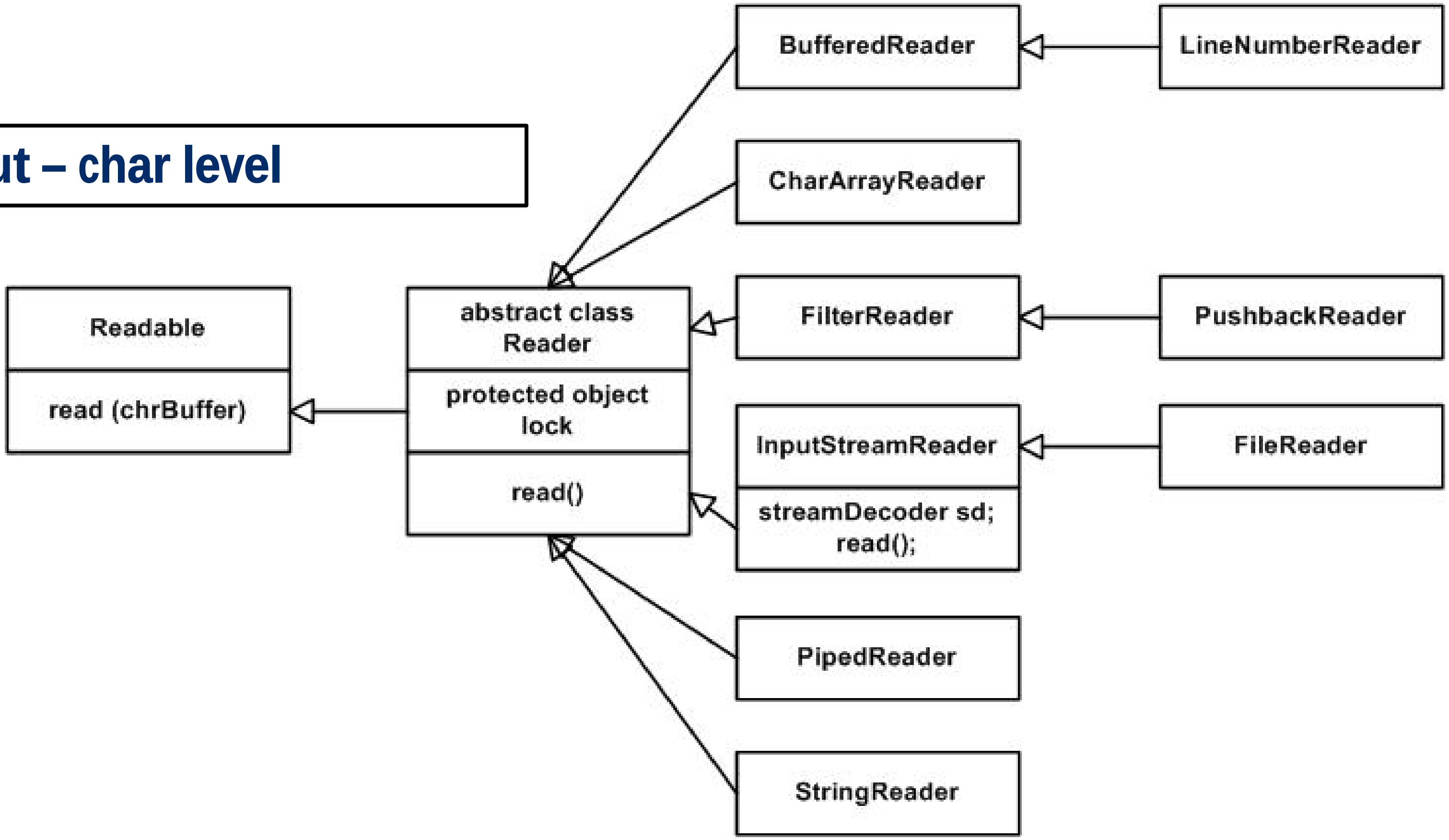
StringBufferInputStream
deprecated

BufferedOutputStream
private static int defaultBufferSize = 8192;
protected volatile byte[] buf
private synchronized int write()

2009-2010 © ism.ase.ro Catalin Boja

Java fundamentals
Files

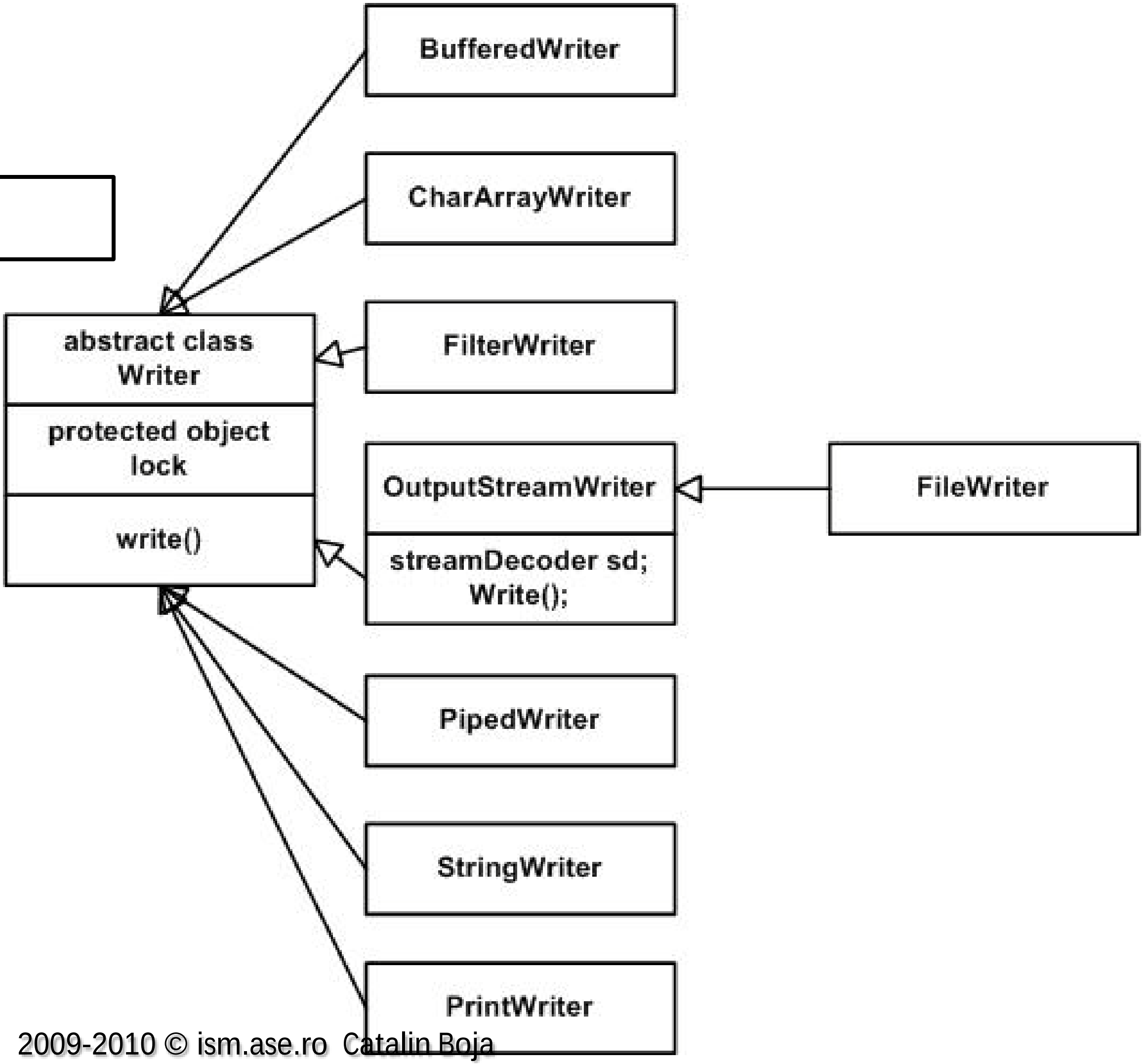
Input – char level



Java fundamentals

Files

Output – char level



Java fundamentals

Files

- **InputStreamReader**, **OutputStreamWriter** - convert bytes to characters and vice versa
- **DataInputStream**, **DataOutputStream** - read and write simple data types
- **ObjectInputStream**, **ObjectOutputStream** – read and write serialized Java objects
- **BufferedInputStream**,**BufferedOutputStream**, **BufferedReader**, **BufferedWriter** - stream filters with buffering

Java fundamentals
Files

- **PrintStream**, **PrintWriter** – prints text;
- **FileInputStream**, **FileOutputStream** - implementations of InputStream, OutputStream
- **FileReader**, **FileWriter** - implementations of Reader, and Writer

Java fundamentals

Files

Class	Extends	Constructor	Methods

File	Object	(String)
		(String,String)
		(File,String)
			createNewFile()
			delete()
			exists()
			isDirectory()
			isFile()
			list()
			mkdir()
			renameTo()

FileWriter	Writer	(File)
		(String)
			cose()
			flush()
			write()

BufferedWriter	Writer	(Writer)	close()
			flush()
			newLine()
			write()

Java fundamentals

Files

Class	Extends	Constructor	Methods

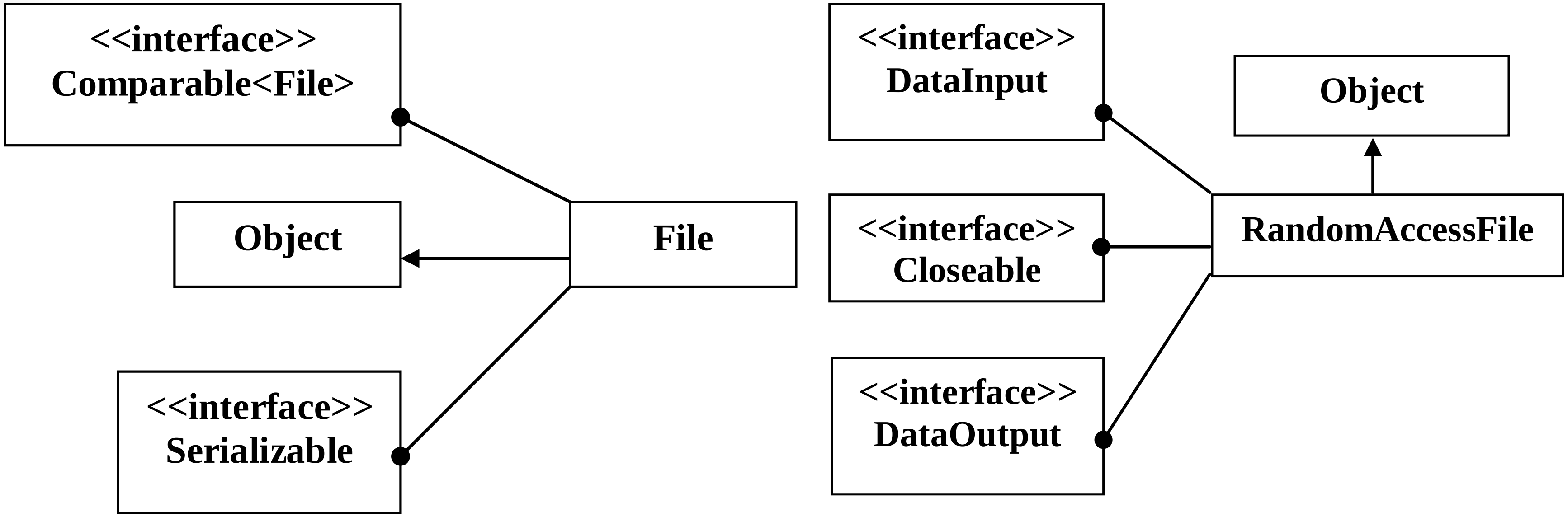
PrintWriter	Writer	(File)
		(String)
		(OutputStream)
		(Writer)
			close()
			flush()
			write()
			print()
			println()
			format()

FileReader	Reader	(File) (String)	read()

BufferedReader	Reader	(Reader)	read()
			readLine()

Java fundamentals

Files



File & RandomAccessFile

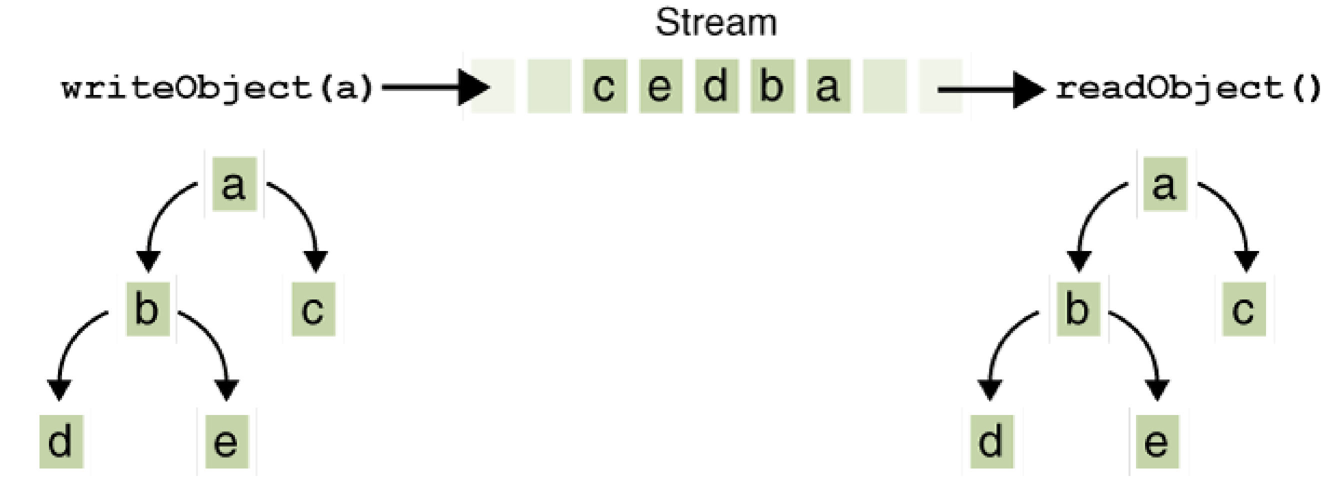
Java fundamentals
Serialization

- Serialization – saves the state of an object
- De-serialization - loads the state of an object
- done by **ObjectInputStream.writeObject()** and **ObjectOutputStream.readObject()**
- the class must implement **Serializable**
- attributes marked as **transient** are not serialized

Java fundamentals

Serialization

Object Graphs



I/O of multiple referred-to objects

Java fundamentals
Serialization

- you can override the mechanism:

```
private void writeObject(ObjectOutputStream os){
os.defaultWriteObject();
//other data
}
```

Java fundamentals
Serialization

- for a serializable class, with a NOT serializable superclass, any inherited instance variable wil be reset to its constructor values;
- static variables are NOT serialized;
- serialization uses a 64-bit hash value, called the Serial Version UID (SUID), to store the version of the class structure:

```
static final long serialVersionUID = 1;
```

Java fundamentals
Threads

- “a thread is a flow of control within a program”;
- a process can have multiple threads;

Defining threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable

class **NewThread** extends **Thread**{
    public void run(){…}
}
	class **NewThread** implements **Runnable**{
	    public void run(){…}
	}

Java fundamentals

Threads

Instantiating threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable

NewThread f = new **NewThread** ();	**NewThread** obf = new **NewThread** ();
	Thread f = new Thread (obf);

Executing threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable
f.start();	f.start();

Java fundamentals
Threads

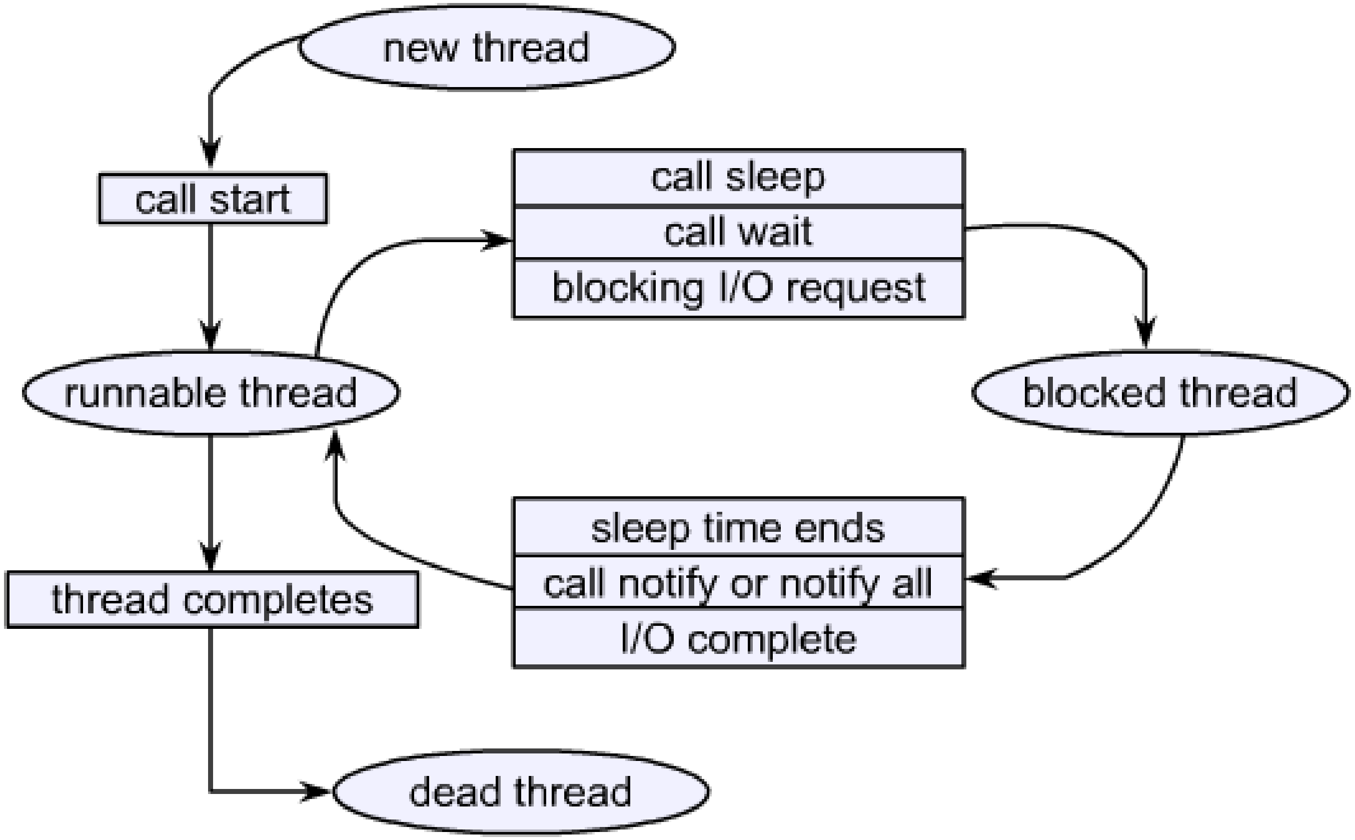
Controlling threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable

public void run(){
this.sleep();
}
	public void run(){
	Thread t = Thread.currentThread();
	...
	t.sleep();
	}

Java fundamentals

Threads



Java fundamentals
Threads

- **new thread** – using new to create an instance
- **runnable thread** – after calling start()
- **blocked thread** – if it is called sleep() or wait()
- **dead thread** – after run() has ended

Java fundamentals
Threads

Threads synchronization:

- concurrency – use same resources;
- cooperation – interchange data;

Controlling threads:

- **wait()**, **notify()** and **notifyAll()** inherited from Object**;**
- **join()** method in Thread class;
- **setPriority()** method in Thread class;
- **yield()** static method in Thread class;
- **sleep()** static method in Thread class;

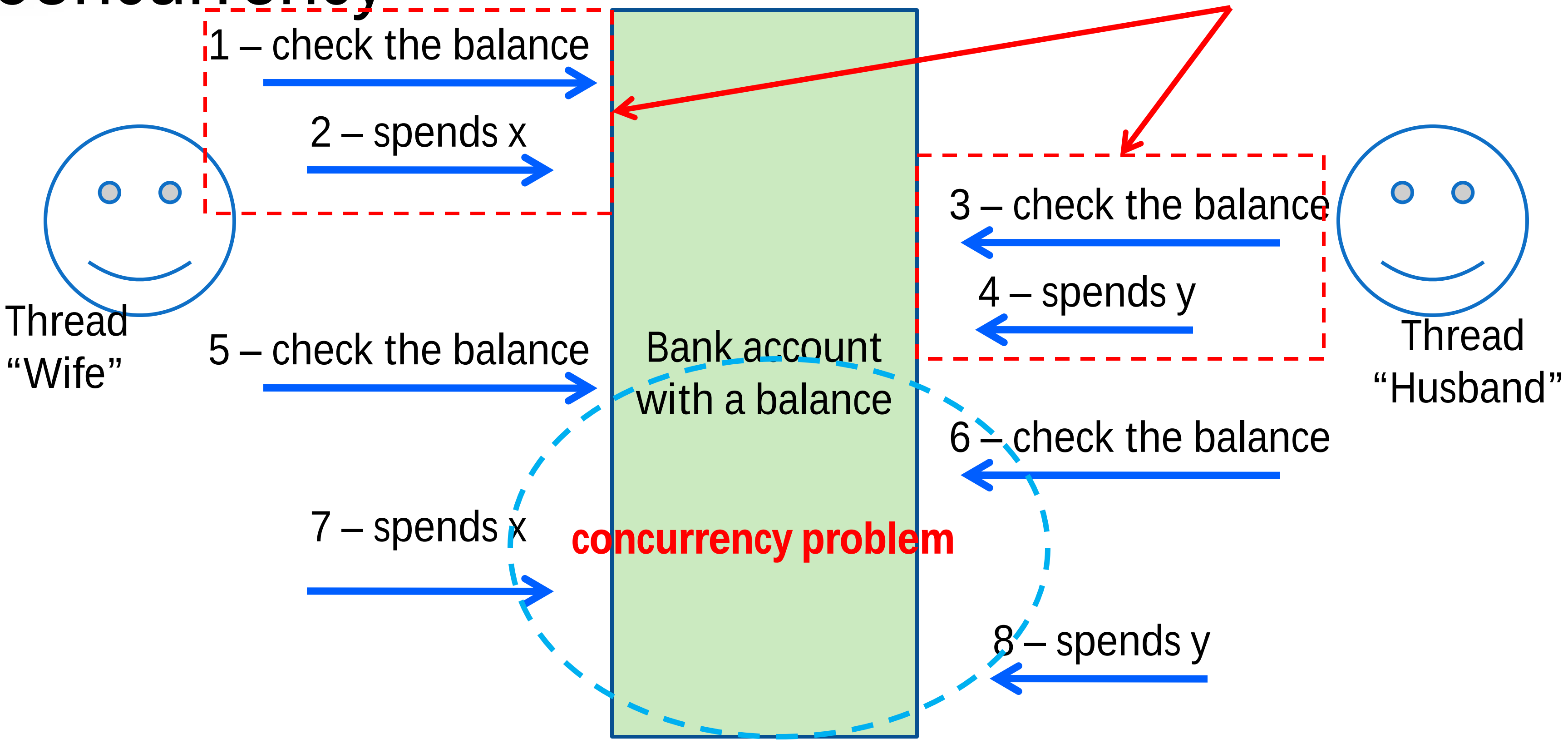
Java fundamentals
Threads

Controlling threads:

- **setPriority() –** sets a priority for the current thread (1->10 or 1->5);
- **join() –** the current threads waits another thread do finish (the current thread joins the other one)
- **yield()** – interrupts the current thread and gives control to another one with the same priority;
- **sleep() -** interrupts the current thread for a period

Java fundamentals
Threads

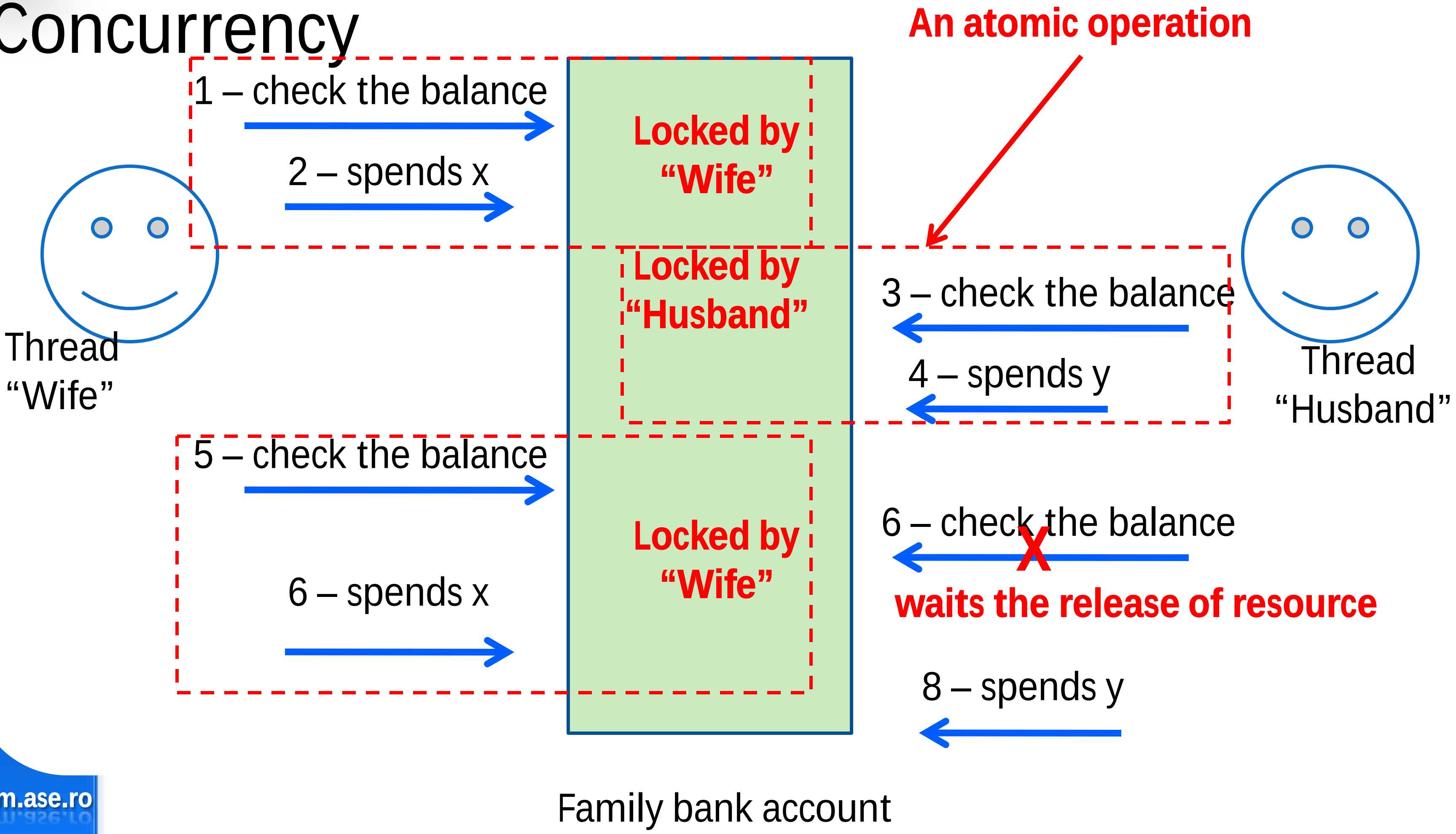
Concurrency



Family bank account

Java fundamentals
Threads

Concurrency



Java fundamentals

Threads

Concurrency

- make syncronized methods or blocks that access common resources;
- each object has one lock (managed by the JVM);
- declare common variables as volatile – modifications are discarded into memory
- use **wait()**, **notify()** and **notifyAll()** inherited from Object;

Java fundamentals
Threads

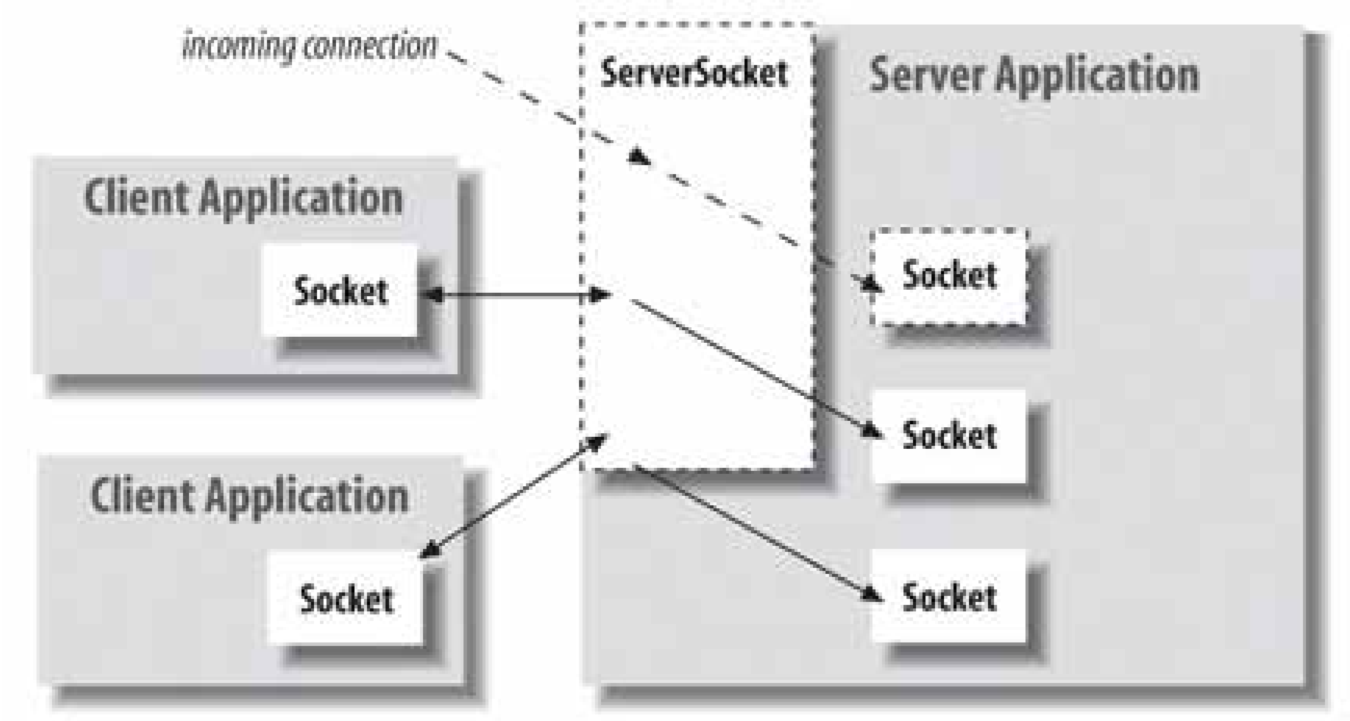
Concurrency

- watch for deadlocked threads;
- **wait()**, **notify()** and **notifyAll()** must be called from a syncronized context;
- common implementation of wait():

```
while (!condition) {
    this.wait();
}
```

Java fundamentals

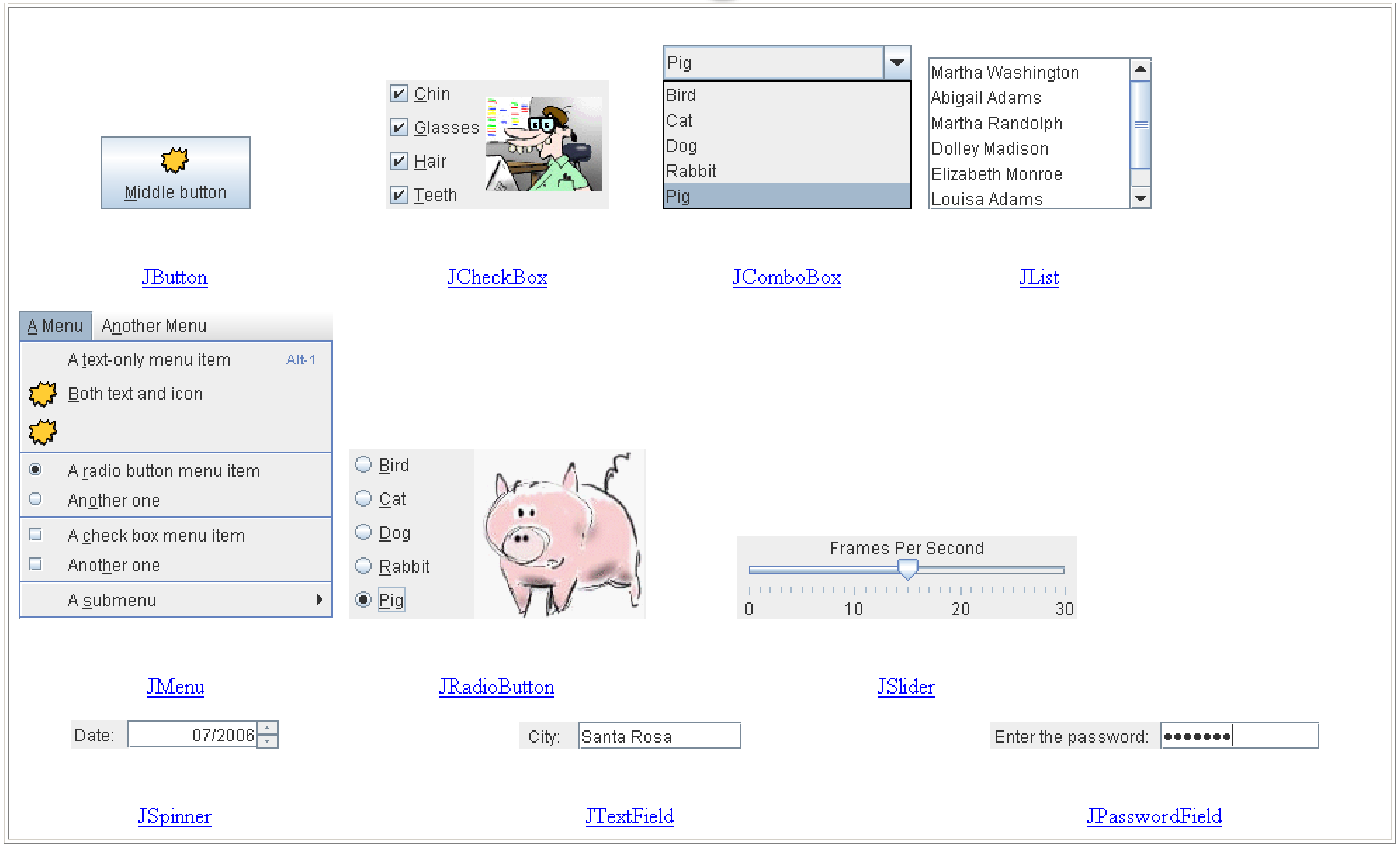
Network I/O



[3]

Java fundamentals

Swing



JButton

JCheckBox

JComboBox

JList

JMenu

JRadioButton

JSlider

Date: 07/2006

City: Santa Rosa

Enter the password: •••••••

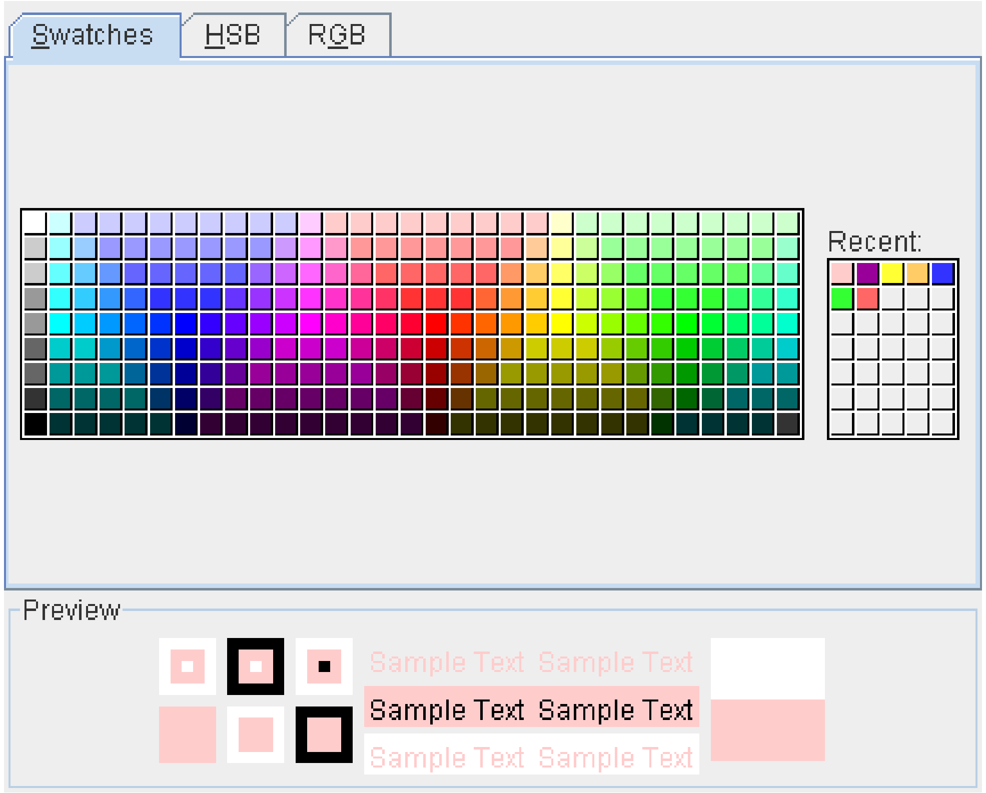
JSpinner

JTextField

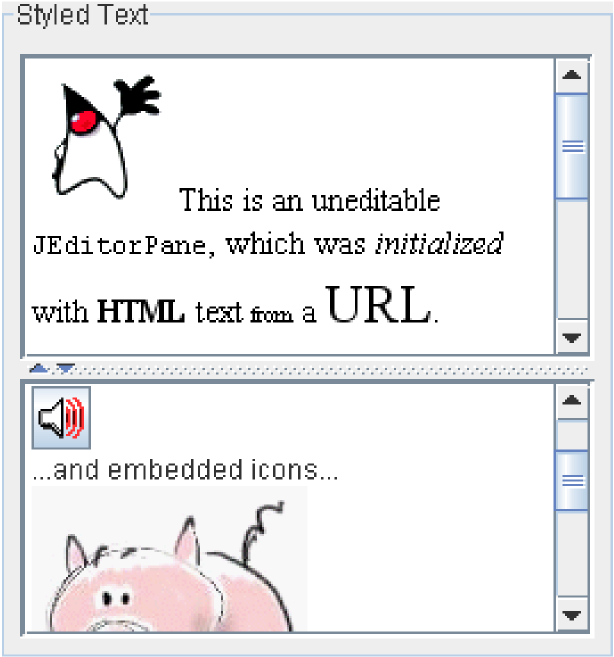
JPasswordField

Java fundamentals

Swing



JColorChooser



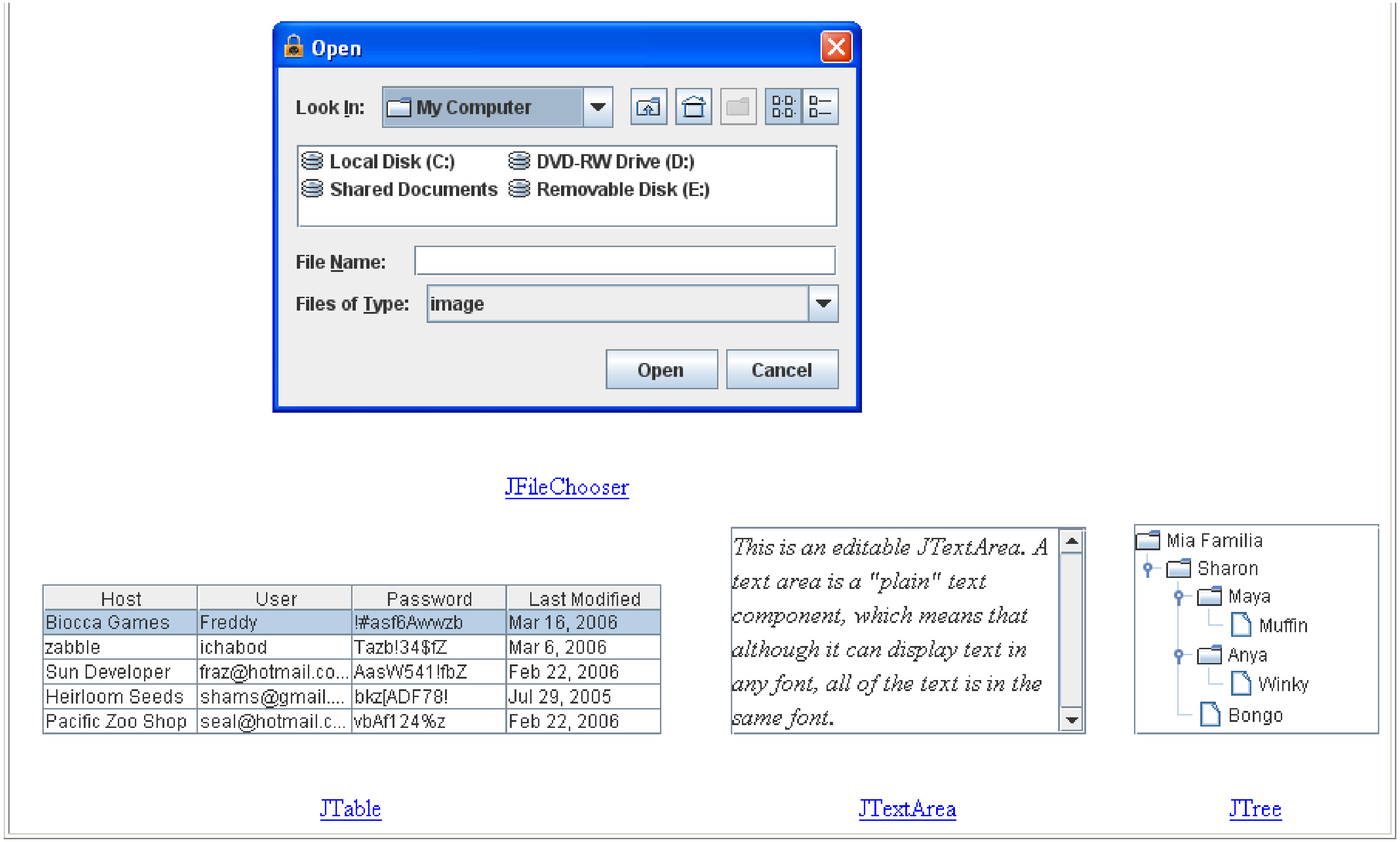
This is an uneditable JEditorPane, which was *initialized* with **HTML** text from a URL.

...and embedded icons...

JEditorPane and JTextPane

Java fundamentals

Swing



JFileChooser

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTable

This is an editable JTextArea. A text area is a "plain" text component, which means that although it can display text in any font, all of the text is in the same font.

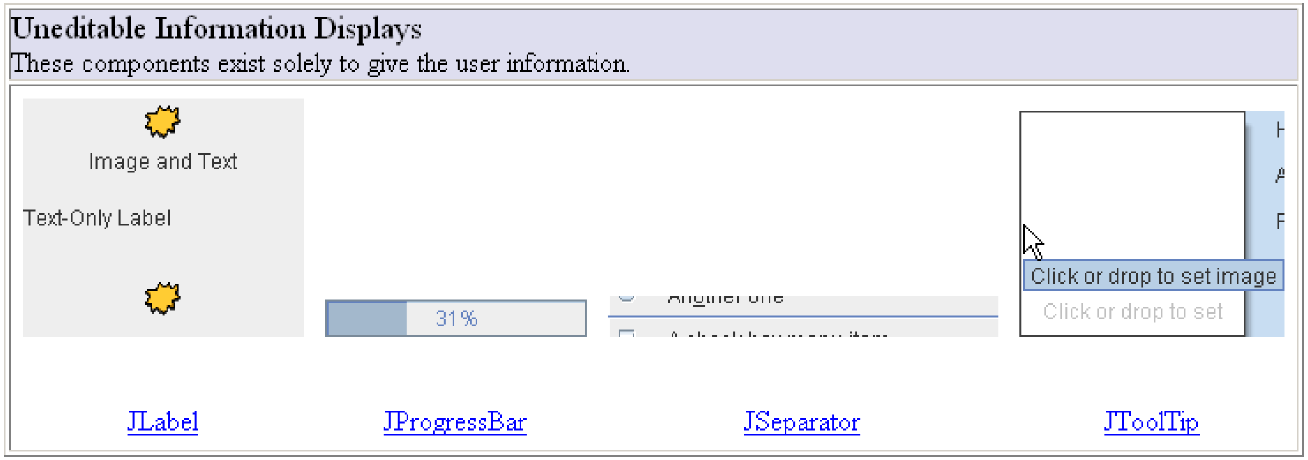
JTextArea

- Mia Familia
  - Sharon
    - Maya
      - Muffin
    - Anya
      - Winky
    - Bongo

JTree

Java fundamentals

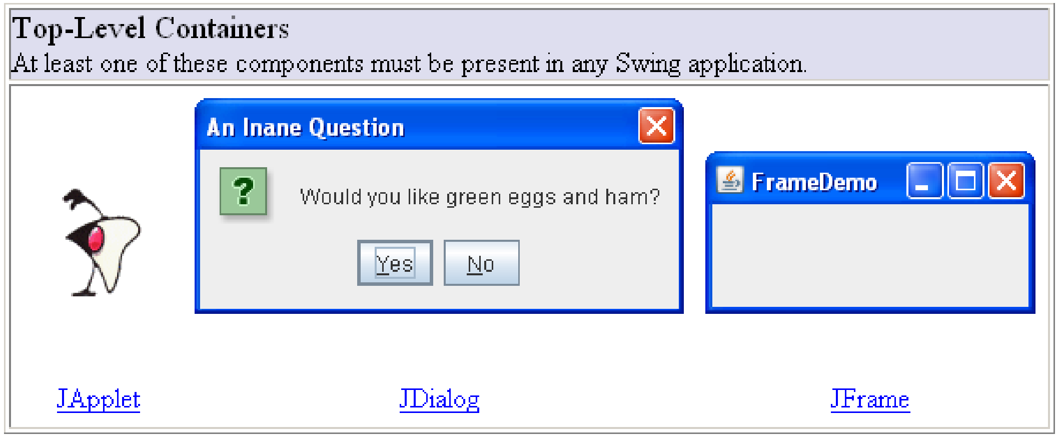
Swing



Uneditable Information Displays

These components exist solely to give the user information.

JLabel JProgressBar JSeparator JToolTip



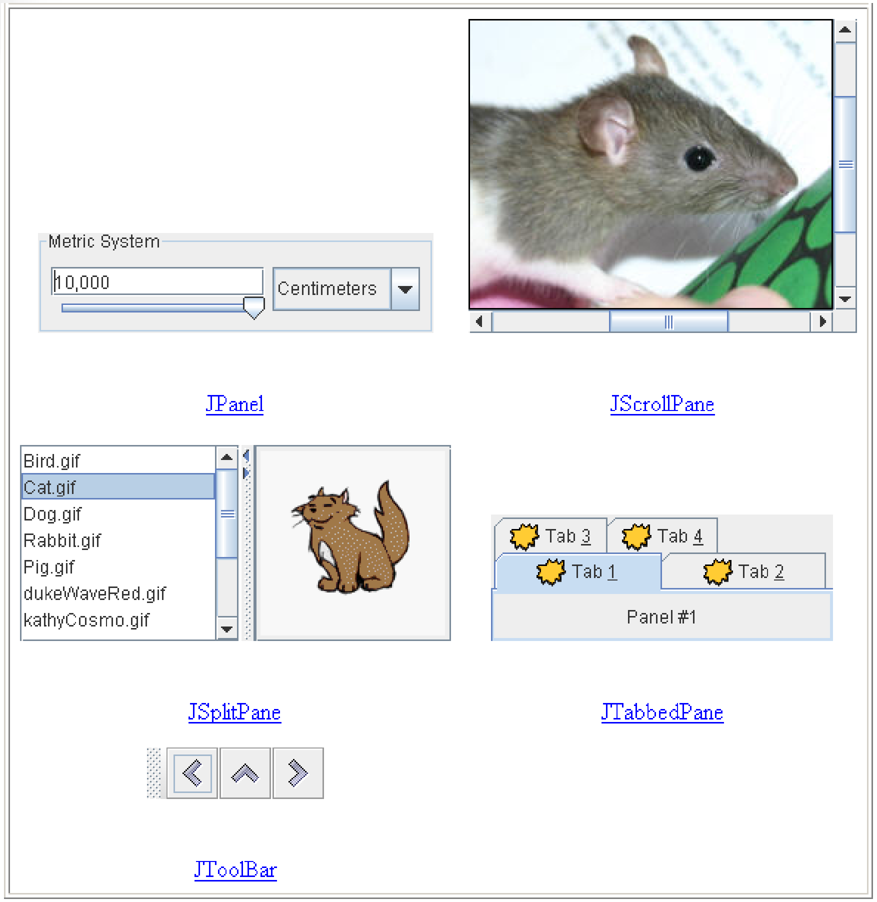
Top-Level Containers

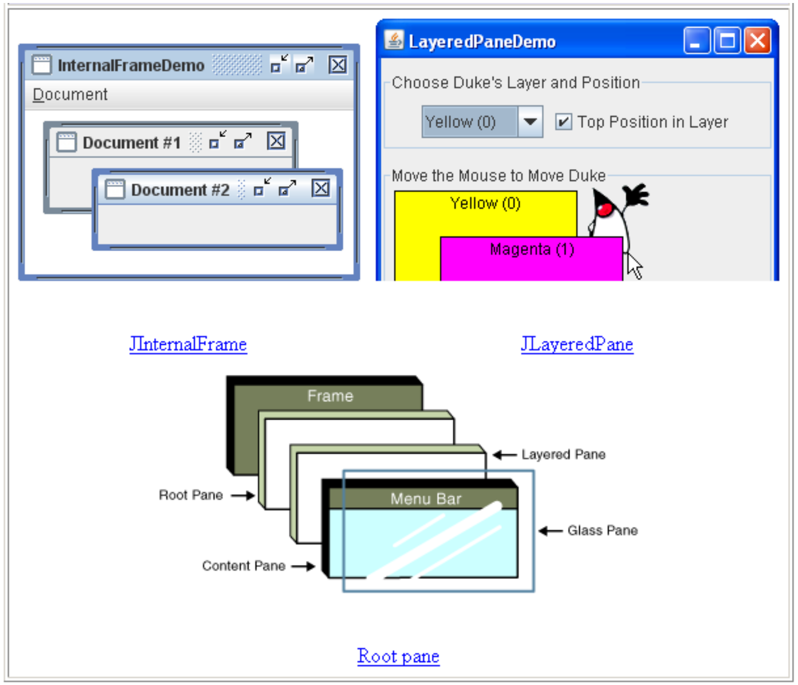
At least one of these components must be present in any Swing application.

JApplet JDialog JFrame

Java fundamentals

Swing





JCA
Java Cryptography Architecture

- SUN JCA - http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
- BouncyCastle - http://www.bouncycastle.org/documentation.html

JCA
Java Cryptography Architecture

- Hash functions – MD5, SHA-1
- Symmetric encryption– DES/AES in ECB and CBC mode
- Asymmetric encryption– RSA
- Digital certificates – X509 v3 (using *keytool* tool or a source code application)

JCA
RSA

Given two primes *p* and *q*, if you have other numbers *n, e*, and *d* such that:

$n = p*q$ and $ed \equiv 1\text{mod}((p - 1)(q - 1))$ then, for a message *m*,

$c = me \text{ mod } n$ - encryption

$m = cd \text{ mod } n$ - decryption

n – modulus

e – *public exponent*

d – private exponent

size of *n* determines how many bits the RSA key

p and *q* need to have a bit length half that of the key size.

[4]