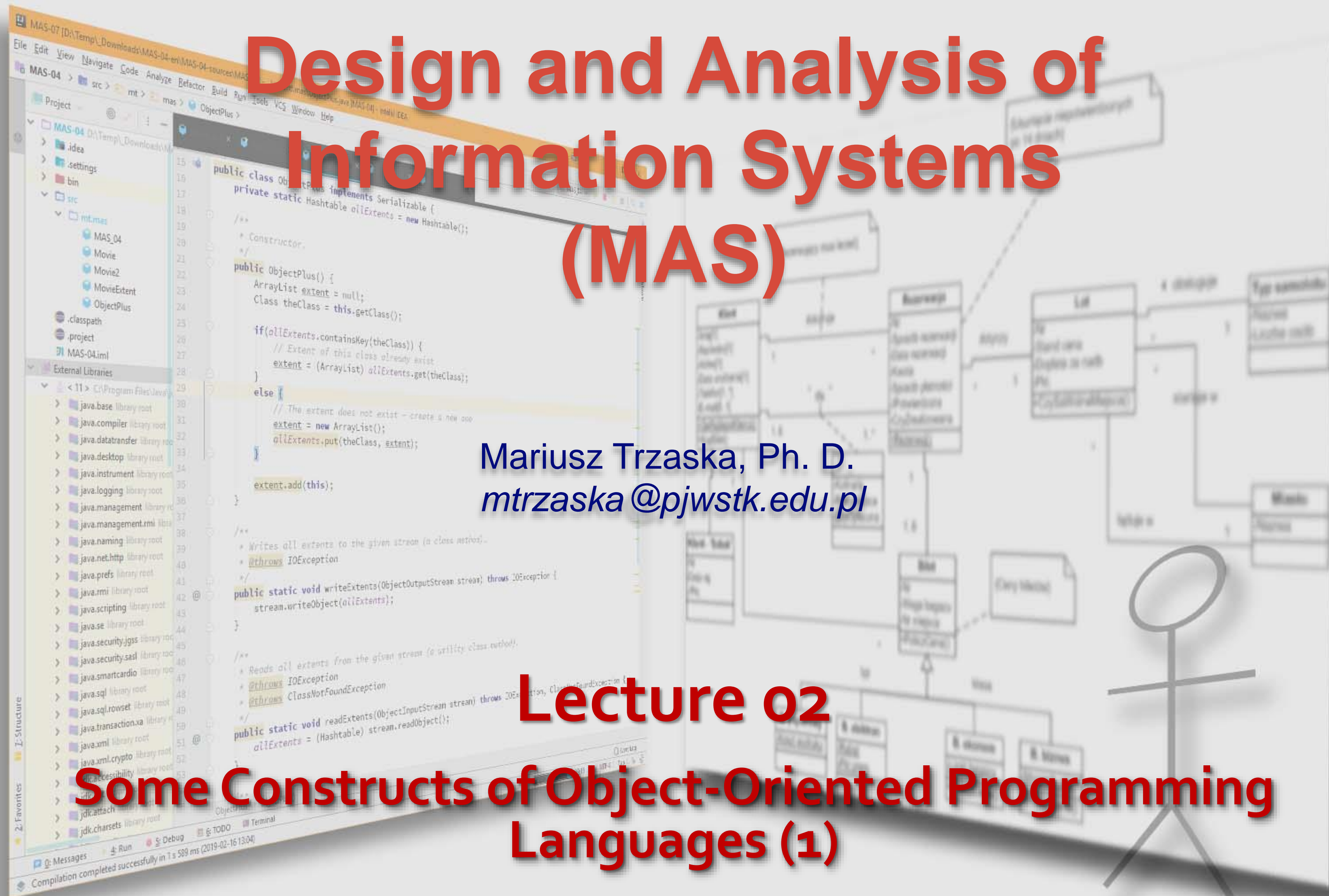
POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY



Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

Lecture 02

Some Constructs of Object-Oriented Programming Languages (1)

http://www.mtrzaska.com/

Outline

- Basic
- Control flow
- Classes
- Interfaces
- Errors handling
- Containers
- IO systems
- Performance
- Summary

These slides make use of the following sources: Thinking in Java (3rd edition) by Bruce'a Eckel'a

http://www.mindview.net/Books/TIJ

Sample programming tasks

- These programming tasks are only intended to guide programming material (learned during previous courses) and their solutions will not be assessed as part of the MAS classes/lectures.
- They can be used during the "Selected constructions of object-oriented programming languages" class.
- Students entering the MAS course should be able to solve most of the tasks.

https://www.mtrzaska.com/mas-programming-tasks/

Basics Types

Primitive type	Size	Minimum	Maximum	Wrapper type
`boolean`	—	—	—	`Boolean`
`char`	16-bit	Unicode 0	Unicode 2^{16}- 1	`Character`
`byte`	8-bit	-128	+127	`Byte`
`short`	16-bit	-2^{15}	$+2^{15}—1$	`Short`
`int`	32-bit	-2^{31}	$+2^{31}—1$	`Integer`
`long`	64-bit	-2^{63}	$+2^{63}—1$	`Long`
`float`	32-bit	IEEE754	IEEE754	`Float`
`double`	64-bit	IEEE754	IEEE754	`Double`
`void`	—	—	—	`Void`

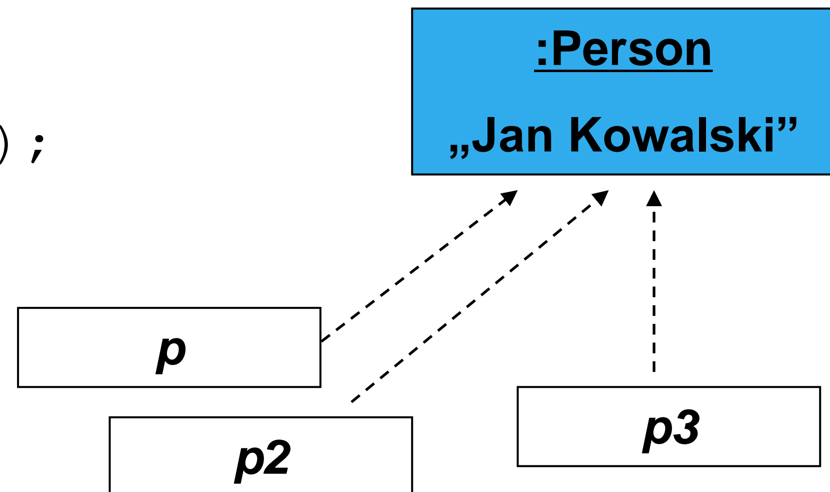
Objects and references

- A basic type

```
int a = 5;
Int b = 7;
```

- An object accessible by a reference

```
Person p = new Person();
Person p2 = p;
Person p3 = p;
```



Wrapper Classes

- `char c = 'x';`
- `Character C = new Character(c);`
- `Character C = new Character('x');`
- Reasons for wrappers?

The Scope

```
{
  int x = 12;
  // Only x is accessible
  {
    int q = 96;
    // x & q are accessible
  }
  // Only x is accessible
}
{
  String s = new String("a string");
} // Scope’s end
```

Classes

```
class ClassName {
  /* Class body */
}

class DataOnly {
        int i;
        float f;
        boolean b;
}
// Creating an object
DataOnly d = new DataOnly( );
```

Classes (2)

- An access to fields (methods and attributes)

```
d.i = 47;
d.f = 1.1f;
d.b = false;
```

- An initialization – a constructor

Primitive type	Default
boolean	false
char	‘\u0000’ (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Methods

- Syntax

```
returnType methodName( /* Parameters*/ ) {
  /* Body */
}
```

- The method takes a *string* and returns an *int*:

```
 int storage(String s) {
   return s.length( ) * 2;
 }
```

- Many parameters for a single method

```
objectName.methodName(arg1, arg2, arg3);
```

A First Program

```java
package com.mt.mas;

import java.time.LocalDate;

public class Main {
    // starting point
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(LocalDate.now());
    }
}
```

Comments

```
/* This is
    a multi-lines
    comment. */

// One-line comment
```

- **Documenting a source code** (*javadoc*)

```
/** The class description */
public class DocTest {

        /** the attribute description */
        public int i;

        /** the method description */
        public void f( ) {}
}
```

Quality of Sources

- Language: native (e.g. Polish), English?
- Names:
  - Classes,
  - Methods,
  - Variables.
- Decomposition into fragments.

Quality of Sources (2)

- Self-documenting/clean code.
  - Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*. ISBN-13: 978-0132350884
- Conventions.
- Formatting sources:
  - Indentation,
  - Curly braces { } – always worth using?
  - Tabs vs spaces.
- Refactoring.

Casting

- Why?

- An example

```
void casts( ) {
   int i = 200;
   long l = (long)i;
   long l2 = (long)200;
}
```

A Control Flow

- If

```
if(Boolean-expression)
   statement
else
   statement
```

- return
- Iteration

```
while(Boolean-expression)
   statement
```

A Control Flow (2)

- do-while

```
do
    statement
while(Boolean-expression);
```

- `break, continue`
- for

```
for(initialization; Boolean-
expression; step)
    statement
```

A Control Flow (3)

- switch

```
switch(integral-selector) {
        case integral-value1:
            statement;
            break;
        case integral-value2:
            statement;
            break;
        // ...

        default: statement;
}
```

Interfaces

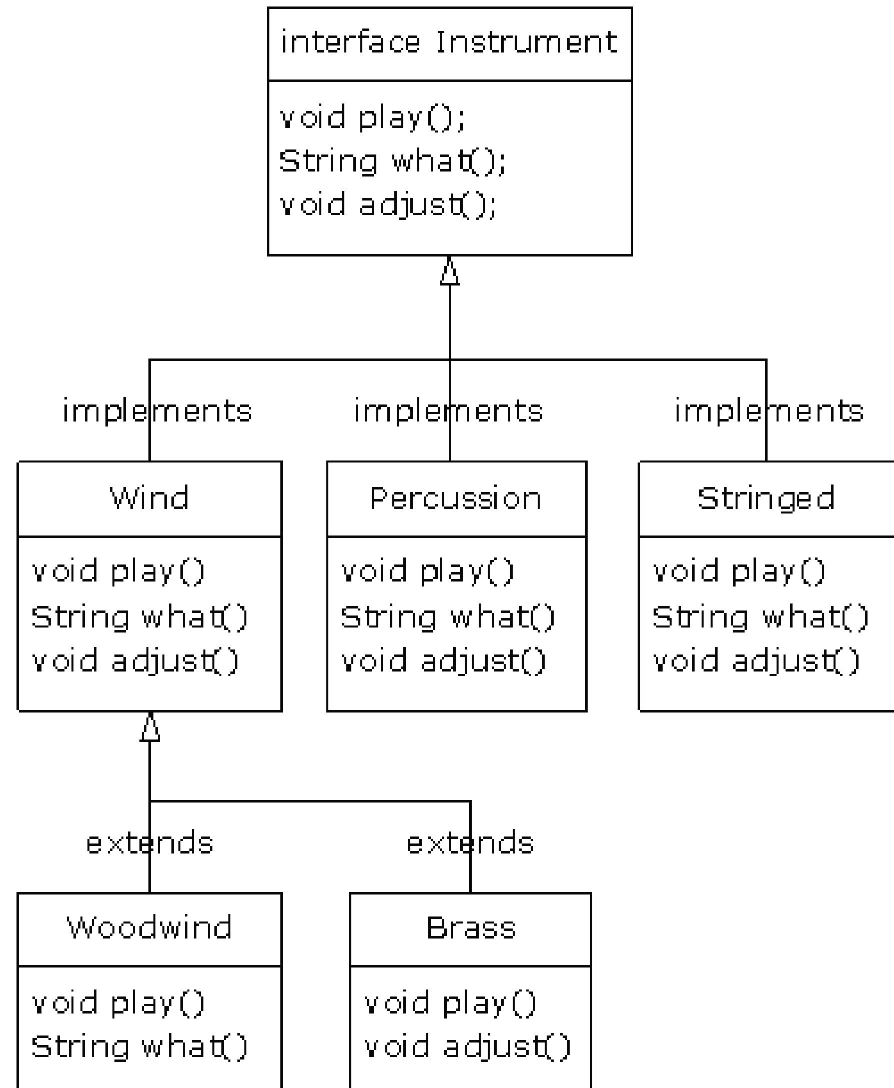
- A purpose
- An example

```
interface Instrument {
   // A constant:
   // static & final
   int I = 5;

   // Declaration only
   // a public method
   void play(Note n);

   String what();

   void adjust();
}
```



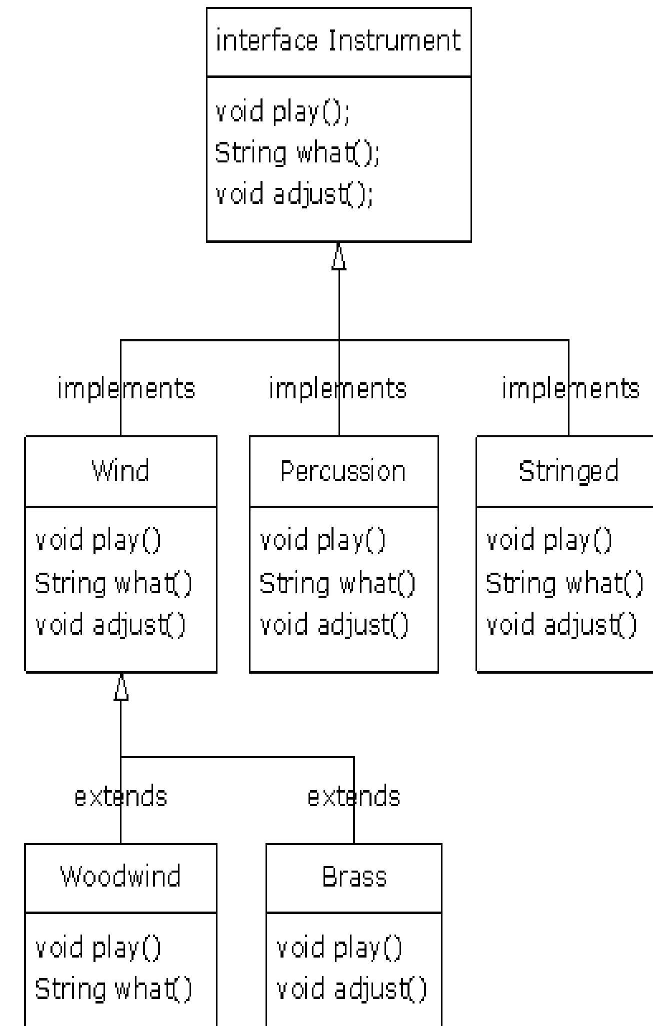
Interfaces(2)

- **Implementation of the interface**

```
class Wind implements Instrument {
  // wind
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
  public String what() { return "Woodwind"; }
}
```



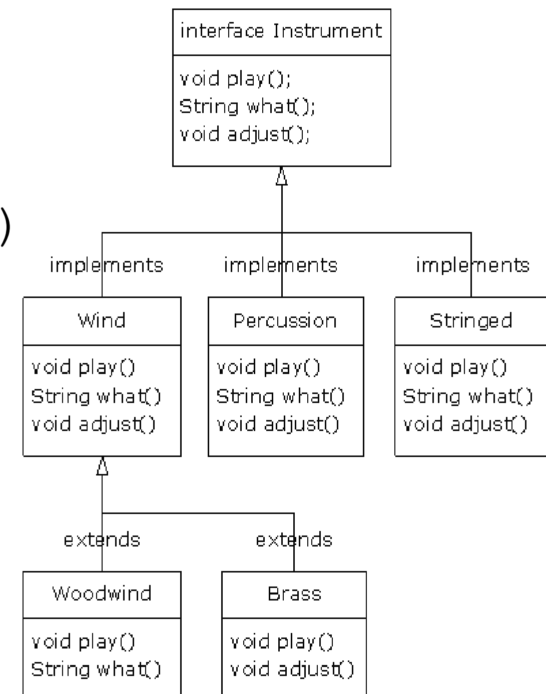
Interfaces (3)

- **Interfaces utilization**

```
Instrument currentInstrument = null;

currentInstrument = new Brass();
currentInstrument.play(1);


currentInstrument = new Woodwind()
currentInstrument.play(1);
```



- A solution with:
  - A base class,
  - Interfaces.

Errors handling

- Errors:
  - Compilation time,
  - Runtime
- Old approaches
  - A special global variable,
  - Returning a special value (which one?)
- The current approach – exceptions
  - Pros
  - Cons (?)

Exceptions

- The class
- Inheritance from the Exception class
- Attributes
- Catching and processing

```
try {
 // a code which may cause n exception
} catch(Type1 id1) {
 // processing of the exception Type1
} catch(Type2 id2) {
 // processing of the exception Type2
}
```

Exceptions (2)

- A method which does not throw exceptions outside but processes them inside

```
void f( ) { // ...
```

- A method which warns that it may throw an exception

```
void f( ) throws TooBigException,DivZeroException
{ //...
```

- *Which approach is better?*

Exceptions (3)

- Catching all exceptions

```
catch(Exception e) {
   System.err.println("Caught an exception");
}
```

- Rethrowing the exception

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
}
```

- Special case: `NullPointerException`

```
person.showName();
```

Exceptions (4)

- Cleaning up with finally

```
try {
  // „Suspicious region”: we expect some
  exceptions:
  // A, B or C
} catch(A a1) {
  // processing for A
} catch(B b1) {
  // processing for B
} catch(C c1) {
  // processing for C
} finally {
  // processed every time!
}
```

Constants' Utilization

- Different needs
  - A constant value (i.e. a iteration number)
  - A classification of different items
    - Old-school solution (not recommended!)

```
public class MainMenu {
  public static final int MENU_FILE   = 0;
  public static final int MENU_EDIT   = 1;
  public static final int MENU_FORMAT = 2;
  public static final int MENU_VIEW   = 3;
}
```

    - A new apporach (*enum* type)

```
public enum MainMenu {FILE, EDIT, FORMAT, VIEW};
```

To be continued…