Object-Oriented Programming

Lecture 1: Introduction to OOP and Java

Sebastian Junges

How can we engineer large software systems?

Create, but also maintain and reuse

How can we create programs to automate ‘real-life’ processes?

Model aspects beyond well-defined mathematical structures

Object-Orientation as a Paradigm

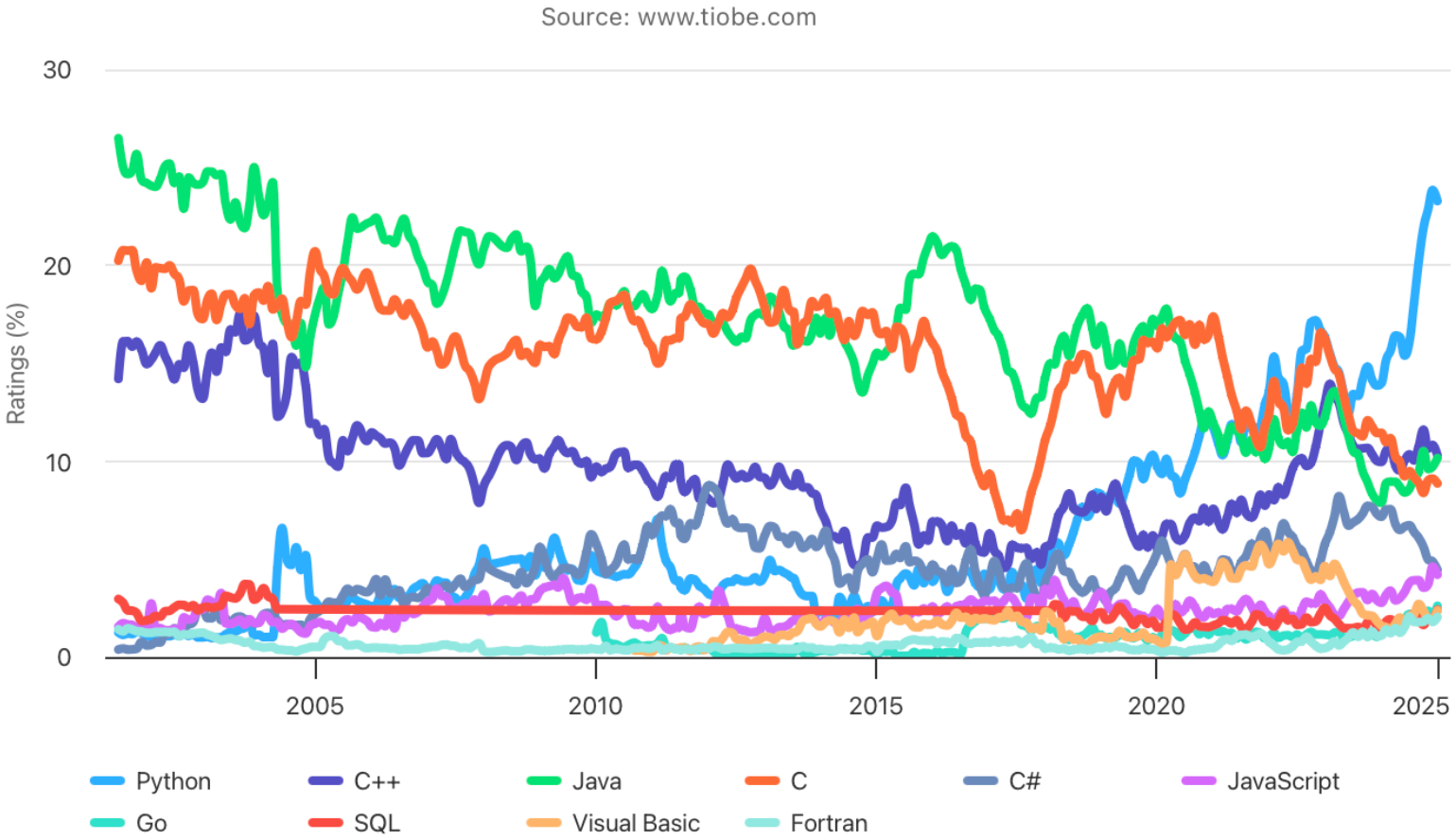
Paradigm = A model or a pattern

- First Object-Orientation in Simula (1960's)
- by Ole-Johan Dahl & Kristen Nygaard
  (who later won the 2001 Turing Award for their contributions)

- **Java**, C++, C#
- Support in Python, Ruby, PHP, VB(.NET), …..
- Many OO-aspects in Go, Rust, JavaScript,

Object orientation is not flawless…,
…but it's ideas have been widely adopted

Programming language ranking

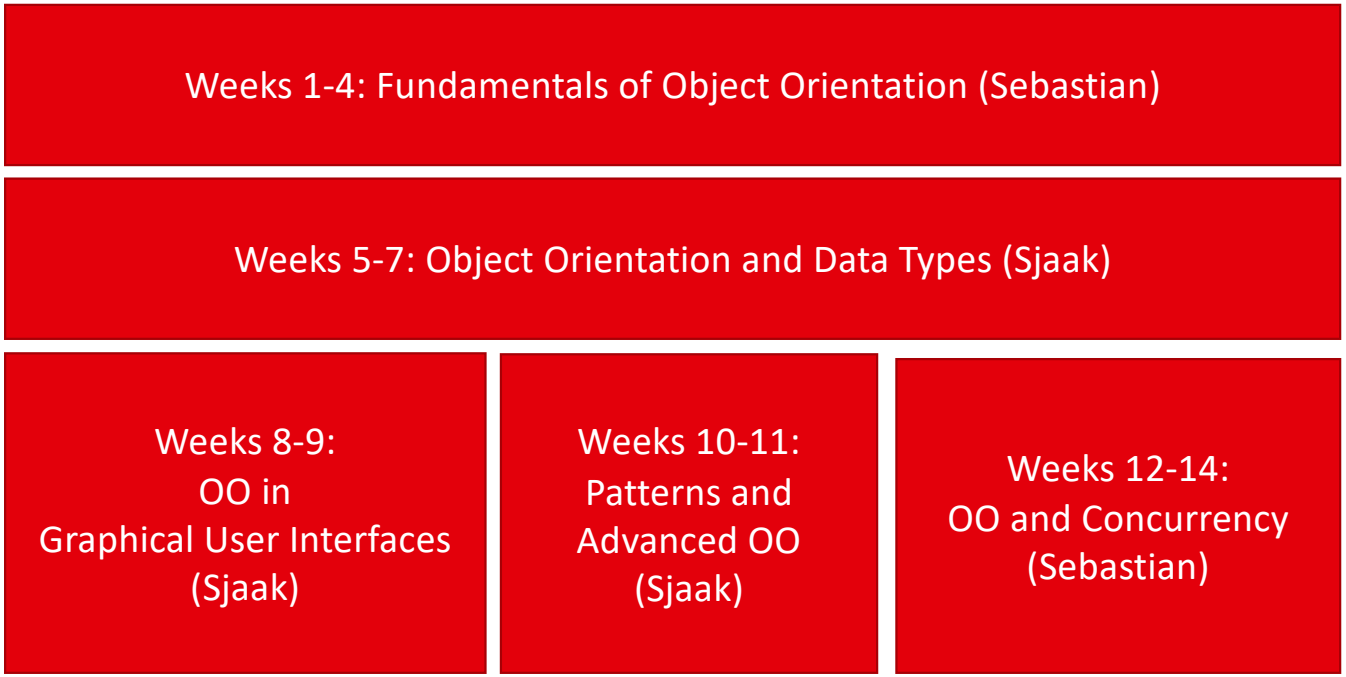


Object-orientation and other courses

Design and programming in OO-style

- Emphasis on algorithms and problem solving
  - 1st Semester Programming course
- OO Design: from requirements to specification
  - Requirements Engineering (NWI-IPC023)
- OO Programming: from specification to implementation
  - **This course**

Course Structure



General Learning Goals

By the end of this course, you can:

- … explain and distinguish the key concepts of object-orientation (Encapsulation, Inheritance, Polymorphism,…)
- … discuss common software design patterns
- … differentiate between methods for multi-threading support
- … develop OO Libraries and (medium-sized) Applications in Java
- … construct Generics
- … use Collections, Streams and other OO data structures
- … work with large OO frameworks, such as for GUIs
- … integrate multi-threading support

Today

Organization & Java

Classes and Objects

Object-Orientation

Classes and Objects in Java

(Arrays)

Organization (1)

Teachers

Sjaak Smetsers

Sebastian Junges

Teaching Assistants

Benedikt Rips

Marck van der Vegt

Homework coordinator & Student Assistants

Maris Galesloot

Organization (2)

Lecture (this)

- Usually monday morning 10:30 – 12:15
- EOS 01.630
- Sometimes on Tuesday due to holidays

Q&A Session / Interactive tutorials

- Tuesday morning 10:30 – 12:15
- HG 00.307

Computer lab (Practicals)

- Thursday morning

Organization (3)

Weekly assignments, **deadlines are strict**! No extensions. **No extensions!**

Assignments are graded using { Fail, Insufficient, Sufficient, Good }

Regulation: **At most 4 fails!**

Fail if

- Incomplete program: program does not compile.
- Essential parts are missing

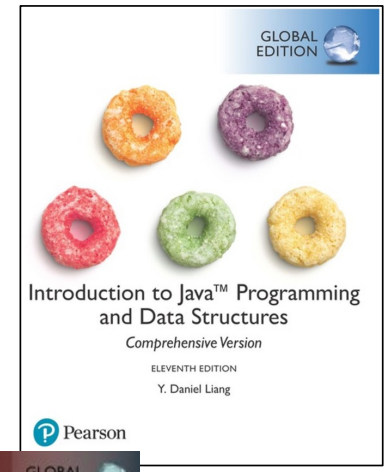
Too many fails: you will be not be admitted to the exam + resit.

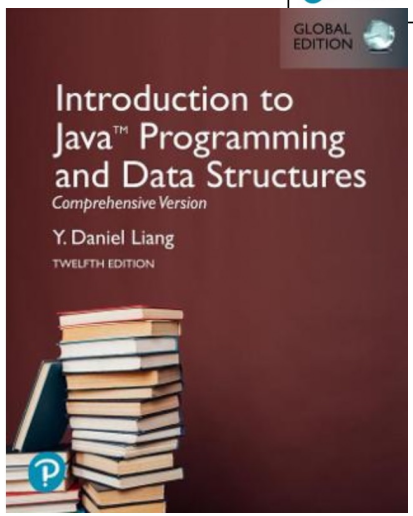
Done OO before? You may opt-out of submitting – Details on Brightspace.

Organization (4)

Book (recommended, not compulsory):

- Intro to Java Programming and Data Structures
- Y. Daniel Liang





We are in this together ☺

- Heterogeneous group with varying programming experience
- Use Q&A session!! Makes homework significantly easier.
- Exam != practical assignments
- Help your fellow students (but no plagiarism)

Hello Java!

Filename: Lecture1.java

```
package sjunges.oolectures.lecture1;

/**
 *  Main class for using data structures developed in Lecture1
 */
public class Lecture1 {
    /**
     * The entry point
     * @param args This argument is required by Java.
     */
    public static void main(String[] args) {
        System.out.println(”Hello world!");
    }
}
```

Every class is part of a package.

```
Compile & Run Lecture1:
Hello World!

Process finished with exit
code 0 / Successful
```

Java programs are run from a main method that looks like this. It is not important what all the code means right now

We can print to the command line using System.out.println(…)

Hello Java!

```
public class Lecture1 {
    public static void main(String[] args) {
        System.out.println(”Hello world!");
    }
}
```

I will simplify matters for readability!

```
Compile & Run Lecture1:
Hello World!

Process finished with exit
code 0 / Successful
```

Java Development Kit

Four phases to running a Java program

- **Edit**
  - store program with the .java file name extension
- **Compile**
  - Use javac (compiler) to create bytecodes from source code; stored in .class files
- **Run** via Java Runtime
  - **Load**: Class loader reads bytecodes from .class files
  - **Verify & Execute**: Java Virtual Machine (JVM) translates bytecodes into machine language

Java API

Java provides class libraries

- Known as Java APIs (Application Programming Interfaces)

To use Java effectively, you must know

- Java programming language
- the extensive library of classes

Using Java API classes instead of writing your own versions can

- improve program performance, because they are carefully written to perform efficiently;
- improve program portability, because they are included in every Java implementation.

Goal Today

- Discuss what classes and objects are
- Explain the main ingredients for a class:
  fields, methods, constructors
- Select access modifiers for encapsulation
- Select whether fields and methods should be static
- (Use arrays in Java)

Today

Organization & Java

Classes and Objects

Object-Orientation

Classes and Objects in Java

(Arrays)

How can we create programs to automate ‘real-life’ processes?

Model aspects beyond well-defined mathematical structures

Objects

In the real world:

- Have a *state*
- Have a *behavior*

In software:

- Have a *state (or properties)* (stored in ***fields***)
- Have a *behavior* (described by ***methods***)

Example Object

State:

- is it closed? `isClosed`
- Is it locked? `isLocked`
- Which material? …

Behavior:

- Close it / Open it
- Lock it / Unlock it (with a key?)
- A door cannot change its material…

Door from: https://commons.wikimedia.org/wiki/File:Door.png

Classes

Classifying objects
describe commonality of sets of similar objects

Describe a blueprint (class) as programmer
let your program “stamp out” any number of instances

Doors from:
https://www.flickr.com/photos/sackton/7580307812/

- Not all doors have locks? Some doors are kind-of-doors?
  Yes, the course takes the complete semester! ☺

Today

Organization & Java

Classes and Objects

Object-Orientation

Classes and Objects in Java

(Arrays)

Object-Oriented Programs

Objects are the building blocks of software systems

- a program is a collection of interacting objects
- objects cooperate to complete a task
- to do this, they communicate by calling (or invoking) each other's methods

Objects are a general concept

Objects may model tangible / concrete things

- school, car, dog

Objects may model conceptual / abstract things

- meeting, date, vehicle

Objects may model processes / tasks

- finding a path through a maze, sorting a deck of cards, handling I/O

Object Oriented Programming

Paradigm: Programming with classes and objects as key building blocks.

- We use Java as a programming language
- the ideas are applicable to other programming languages
- Likewise, many (‘real’) languages have notions like verbs, nouns, ...

Furthermore: many languages are support multiple paradigms;

hard to give a clear and consise definition of what OO means in first lecture.

Today

Organization & Java

Classes and Objects

Object-Orientation

Classes and Objects in Java

(Arrays)

Example Object

Let's write code to describe doors!

State:

- is it closed? `isClosed`
- Is it locked? `isLocked`
- Which material? …

Behavior:

- Close it / Open it
- Lock it / Unlock it (with a key?)
- A door cannot change its material…

Door from: https://commons.wikimedia.org/wiki/File:Door.png

We start with the state of doors and the ability to create objects of them.

The Door Class (only the fields)

```
package sjunges.oolectures.lecture1;

/**
 * This comment describes the Door class
 */
public class Door {
    /** True iff the door is closed. */
    boolean isClosed;
    /** True iff the door is locked. */
    boolean isLocked;
    /** A textual description of the material */
    final String material;

    …
}
```

Fields require a type and a name.
Final fields never change their value after being initialized

Example types: boolean, int, String

The Door Class (fields & constructor)

```
public class Door {
    boolean isClosed;
    boolean isLocked;
    final String material;

    /**
     * Constructs a closed, unlocked door with the specified material
     * @param materialForDoor Textual description of the material
     */
    public Door(String materialForDoor) {
        isClosed = true;
        isLocked = false;
        material = materialForDoor;
    }
}
```

Objects are created from classes by calling the Constructor: A method with the same name as the class.

Using the **Door** Class in the main method (1)

```
public class Lecture1 {
    public static void main(String[] args) {
        Door d1 = new Door("wooden");
        Door d2 = new Door("iron");
        System.out.println(d1.material);
        System.out.println(d2.material);
    }
}
```

With constructors ('new') we can create instances of a class (objects).

```
Compile & Run Lecture1:
wooden
iron

Successful
```

Using the **Door** Class in the main method (2)

```
public class Lecture1 {
    public static void main(String[] args) {
        Door d1 = new Door("wooden");
        Door d2 = new Door("iron");
        d1.material = "steel";
    }
}
```

You cannot write to final fields

```
Compile & Run Lecture1:

Error
cannot assign a value to final variable material
```

Using the **Door** Class in the main method (2)

```
public class Lecture1 {
    public static void main(String[] args) {
        Door d1 = new Door("wooden");
        Door d2 = new Door("iron");
        System.out.println(d1.isClosed);
        System.out.println(d2.isClosed);
        d1.isClosed = false; // this is problematic. Do not do this.
        System.out.println(d1.isClosed);
        System.out.println(d2.isClosed);
    }
}
```

```
Compile & Run Lecture1:
true
true
false
true

Successful
```

You can access fields of an object with objectName.fieldName

So far, we have only worked with the state of doors.

Using the state directly allows us to use doors to exhibit impossible behavior.

Encapsulation (first round definition)

Only change the state of an object via its behavior!

The door should only be opened if the door is not locked.
This should always be ensured by the programmer of Door!

We now add behavior to doors.

Methods! (1)

```
public class Door {
    boolean isClosed;
    boolean isLocked;
    String material;


    …

    /** Open the door. Impossible if the door is locked. */
    void open() {
        if (!isLocked) {
            isClosed = false;
        }
    }

    /** Close the door. Impossible if the door is locked. */
    void close() {
        if (!isLocked) {
            isClosed = true;
        }
    }
}
```

Methods! (2)

```
public class Door {
    /**
     * Lock the the door, but only if you provide the right key.
     * @param key 42 is the right key.
     */
    void lock(int key) {
        // Apparently, all doors have the same door to keep the slides simple.
        if (key == 42) {
            isLocked = true;
        }
    }

    /**
     * Unlock the door, but only if you provide the right key.
     * @param key 42 is the right key.
     */
    void unlock(int key) {
        if (key == 42) {
            isLocked = false;
        }
    }
}
```

Using **Door** Methods

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    System.out.println(d1.isClosed);
    d1.lock(42);
    d1.open();
    System.out.println(d1.isClosed);



}
```

```
Compile & Run Lecture1:
true
true


Successful
```

Using **Door** Methods

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    System.out.println(d1.isClosed);
    d1.lock(42);
    d1.open();
    System.out.println(d1.isClosed);
    d1.unlock(41);
    d1.open();
    System.out.println(d1.isClosed);


}
```

```
Compile & Run Lecture1:
true
true
true


Successful
```

Methods can be invoked with objectName.methodName(…).
Methods change fields of (only) the object objectName

Using **Door** Methods

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    System.out.println(d1.isClosed);
    d1.lock(42);
    d1.open();
    System.out.println(d1.isClosed);
    d1.unlock(41);
    d1.open();
    System.out.println(d1.isClosed);
    d1.isClosed = false; // No No No
    System.out.println(d1.isClosed);
}
```

```
Compile & Run Lecture1:
true
true
true
false

Successful
```

Access modifiers

```
public class Door {
    private boolean isClosed;
    private boolean isLocked;
    private String material;
    …

    /** Open the door. Impossible if the door is locked. */
    public void open() {
        if (!isLocked) {
            isClosed = false;
        }
    }




}
```

Access modifiers

```
public class Door {
    private boolean isClosed;
    private boolean isLocked;
    private String material;
    …

    /** Open the door. Impossible if the door is locked. */
    public void open() {
        if (!isLocked) {
            isClosed = false;
        }
    }

    /** Close the door. Impossible if the door is locked. */
    public void close() {
        if (!isLocked) {
            isClosed = true;
        }
    }
}
```

Rule of thumb: Make fields private and only the necessary behavior public!

Using the **Door** Class with access modifiers

```java
public class Lecture1 {
    public static void main(String[] args) {
        Door d1 = new Door("wooden");
        d1.isClosed = false; // this is problematic and no longer compiles :-)
        System.out.println(d1.isClosed); // neither does this :-(
    }
}
```

```
Compile & Run Lecture1:

Error
isClosed has private access in package.Door
```

Access modifiers in Java

Lecture 3

From:	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
(default)	Y	Y	N	N
private	Y	N	N	N

Using **Door** Methods

```
public class Door {
    public boolean getIsClosed() {
        return isClosed;
    }

    public boolean getIsLocked() {
        return isLocked;
    }
}
```

Methods can have a return-type different than void.

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    System.out.println(d1.getIsClosed());

    …
}
```

Use public getters to enable read-access to private fields

```
Compile & Run Lecture1:
true

Successful
```

Methods for conceptual behavior

```
public String toString() {
    // This code will not win a beauty contest,
    // but it is perfectly fine for Lecture 1.
    String result =  "The " + material + " door is ";
    if (isLocked) {
        result = result + "locked";
    } else {
        result = result + "unlocked";
    }
    result = result + " and ";
    if (isClosed) {
        result = result + "closed.";
    } else {
        result = result + "open.";
    }
    return result;
}
```

Methods can also be used
to support the ease-of-use of a class

Using toString

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    d1.lock(42);
    System.out.println(d1.toString());
    d1.unlock(42);
    d1.open();
    System.out.println(d1.toString());
    // And with a touch of magic
    // explained in another lecture:
    System.out.println(d1);
}
```

```
Compile & Run Lecture1:
The wooden door is locked and closed.
The wooden door is unlocked and open.
The wooden door is unlocked and open.

Successful
```

Methods for Auxiliary Functions

```
public class Door {

    public void lock(int key) {
        if (isKeyCorrect(key)) {
            isLocked = true;
        }
    }

    public void unlock(int key) {
        if (isKeyCorrect(key)) {
            isLocked = false;
        }
    }

    private boolean isKeyCorrect(int key) {
        return (key == 42);
    }
}
```

Private methods help to structure the code and to avoid repetition

We have created doors with a state and behavior, as well as the ability to construct them.

Let’s recap more generally!

The syntax of a class definition

```
// file: <ClassName>.java
<accessModifier> class <ClassName>
{
// Fields
    <accessModifier> <type> <fieldName>;
    <accessModifier> <type> <fieldName>;

// Constructors
    <accessModifier> <classname>(<type> <argName>, …, <type> <argName>)
    {
        …
    }

// Methods
    <accessModifier> <type> <methodName>(<type> <argName>, …, <type> <argName>)
    {
        …
    }
}
```

Two sorts of types

Two sorts of types:

- Primitive (int, boolean, …)
- Reference (Classes, e.g., Door, Lecture1, Greeter,…)

Primitive types:

- int (4), long (8), short (2), byte (1)
- double (8), float (4)
- char, boolean
- Operations like +, -, * supported

Reference types:

- classes: every class is a (reference) type.

Special type: `String`

- Belongs to "`java.lang.*`" API
- Not primitive!
- Support for concatenation via: +

One more ingredient: static fields and methods

Static fields by example

```
public class Door {

    …
    /** A unique id for every door */
    private int id;
    /** Counter for the ids */
    private static int nextId = 0;

    public Door(String materialForDoor) {
        id = nextId;
        nextId++;
        …
    }

    public int getId() {
        return id;
    }
}
```

static fields are independent of the object

Static fields demonstrated

```
public static void main(String[] args) {
    Door d1 = new Door("wooden");
    Door d2 = new Door("iron");
    System.out.println(d1.getId());
    System.out.println(d2.getId());
    Door d3 = new Door("wooden");
    System.out.println(d1.getId());
    System.out.println(d3.getId());
}
```

```
Compile & Run Lecture1:
0
1
0
2

Successful
```

The keyword static

- Static fields: *exist once per class.*
- Static methods: *are independent of any object.*
  Cannot access instance fields or methods.

  Examples for static methods:
  - `String.valueOf(int x)`
  - `static main(...)`

Wrap-up: Classes

Methods may change the values of the fields

Constructors (special methods) invoked implicitly (via **`new`**) to create an instance (object) of a class

Fields (instance variables) typically initialized by the constructor.

Static fields (class variables) shared by all objects of a class

Static methods (functions) cannot access (instance) fields.

Today

Organization & Java

Classes and Objects

Object-Orientation

Classes and Objects in Java

(Arrays)

Arrays

An array is a special kind of object

Think of as an ordered list of variables of the same type

Initializing an array: `double[] temperature = new double[3];`

or `double[] temperature = new double[3] { 3.3, 15.8, 9.7 };`

or even simpler = `double[] temperature = { 3.3, 15.8, 9.7 };`

The attribute `length`

As an object an array has one public field (an *attribute)*

- name `length`
- Contains number of elements in the array
- value cannot be changed (final)

For-loops in Java

```
public static void main(String[] args) {
        int[] array = { 87, 68, 94, 100, 83, 78 };
        int total1 = 0;
        int total2 = 0;

        // add each element's value to total. Read-only access, readable
        for ( int number : array ) {
            total1 += number;
        }
        // add each element's value to total. less readable
        for (int j = 0; j < array.length; j++) {
            total2 += array[j];
        }

        System.out.println( "Total of array elements: " + String.valueOf(total1));
        System.out.println( "Total of array elements: " + String.valueOf(total2));
    }
```

Next lecture: 3/2

Separation of Concerns & Interfaces

Q&A on Tuesday!