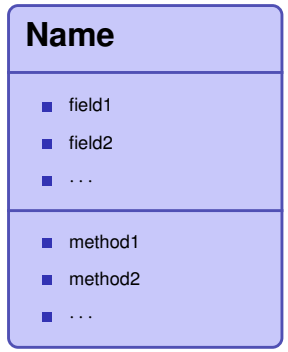
12. Java Classes

Classes, types, objects, declaration, instantiation, constructors, encapsulation, static fields and methods

Classes - Technical

A class is an entity with a *name* that contains *data* and *functionality*

- A class defines a new *data type*.
- *Data* consists of variables that we call *fields* or *attributes*.
- *Functionality* consists as *methods* that are defined within the class.
- Classes are (typically) separate `.java` files with the same name.

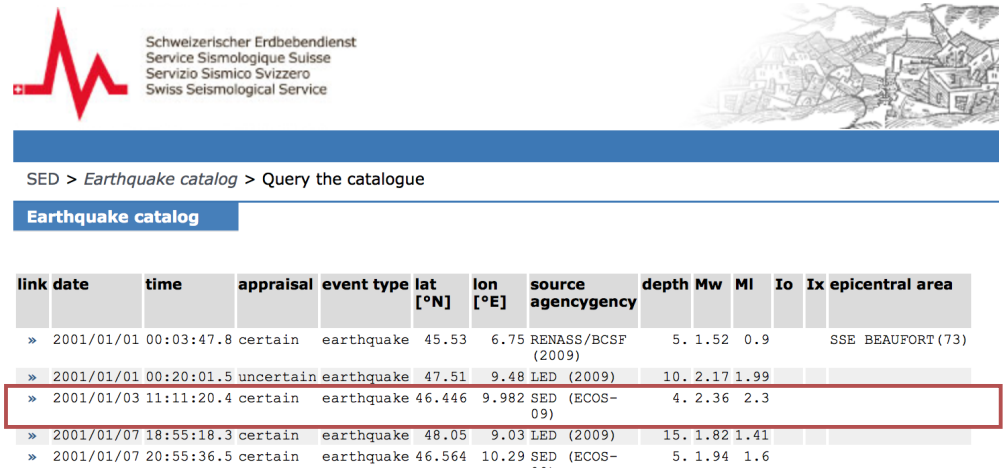


Classes - Conceptual

Classes facilitate to *bundle* the data that *belongs together* content wise.

Classes provide *functionality* that allows to perform *queries* based on the data or *operations* on the data.

Example: Earthquake catalog

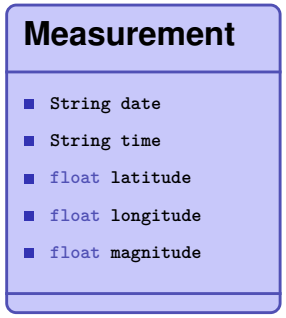


Class for measurement - first try

date	time	appraisal	event type	lat [°N]	lon [°E]	source agencygency	depth	Mw
2001/01/03	11:11:20.4	certain	earthquake	46.446	9.982	SED (ECOS-09)	4.	2.36

File `Measurement.java`

```
public class Measurement {

    String date;
    String time;

    float latitude;
    float longitude;

    float magnitude;
}
```



Objects: Instances of Classes

Classes describe the structure of objects, like a *blueprint*

⇒ Comparable with the *header* of the CSV.

Objects are instantiated according to the blueprint and will contain values

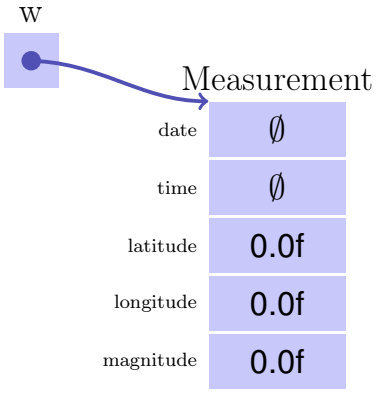
⇒ Comparable with the individual *data-rows* in the CSV.

Object Instantiation: The Keyword `new`

Variable "w" of type "Measurement"

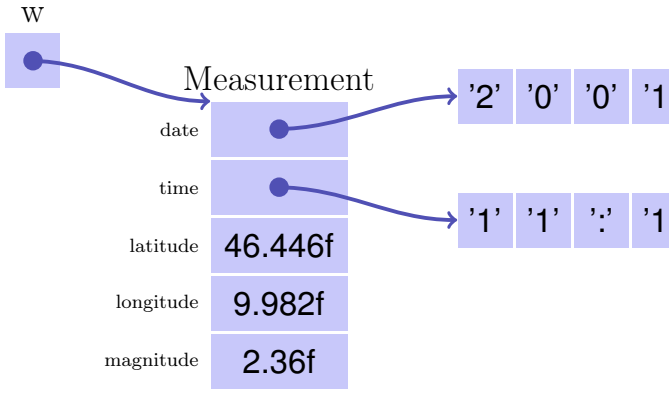
```
Measurement w;

w = new Measurement();
```

Instantiation of an object of type Measurement



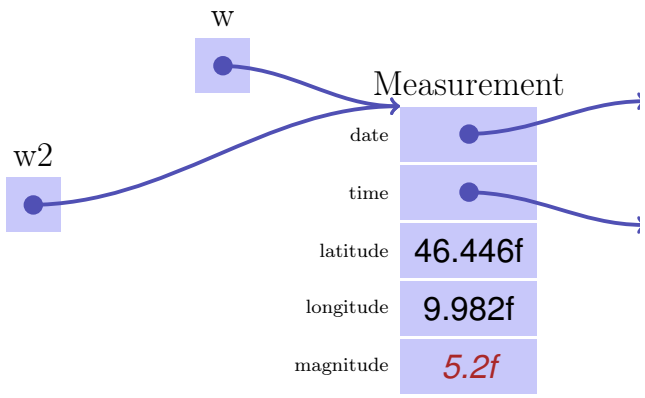
De-referencing: Accessing Fields

```
Measurement w;

w = new Measurement();

w.date = "2001/01/03";
w.time = "11:11:20.4";
w.latitude  = 46.446f;
w.latitude  = 9.982f;
w.magnitude = 2.36f;
```



Objects are Reference-Types: Aliasing

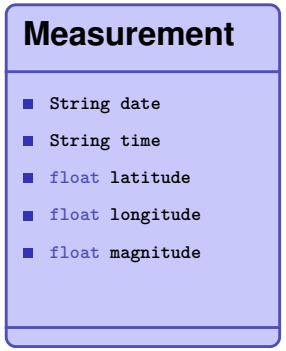
```
Measurement w;

w = new Measurement();

Measurement w2 = w;

w2.magintude = 5.2f;

Out.println(w.magnitude);
```



Wait a second! ...

Classes facilitate to *bundle* the data that *belongs together* content wise.

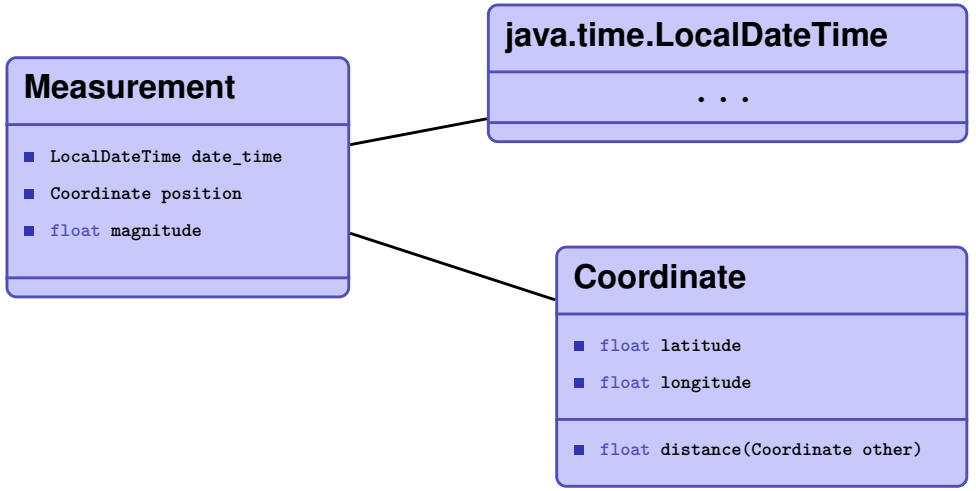
Good Class Design?



We can do better!

- Date and Time belong together in a separate class: Java already offers this: `java.time.LocalDateTime`
- Latitude and longitude belog in their own data type `Coordinate`.

Class Design - second try

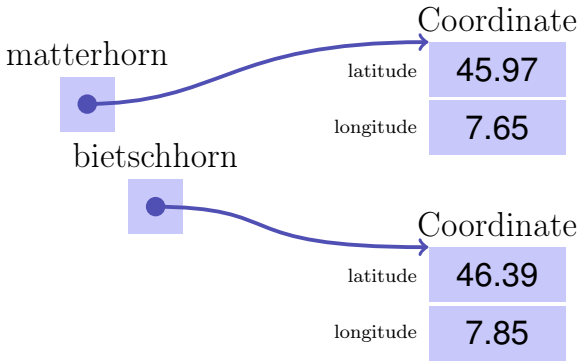


Methods in Classes

```
public class Coordinate {

    float latitude;
    float longitude;

    /**
     * Computes the distance to the provided
     * coordinate 'other' in kilometers.
     */
    float distance(Coordinate other){
        float dl = this.latitude - other.latitude;
        // complete this as exercise ...
    }
}
```

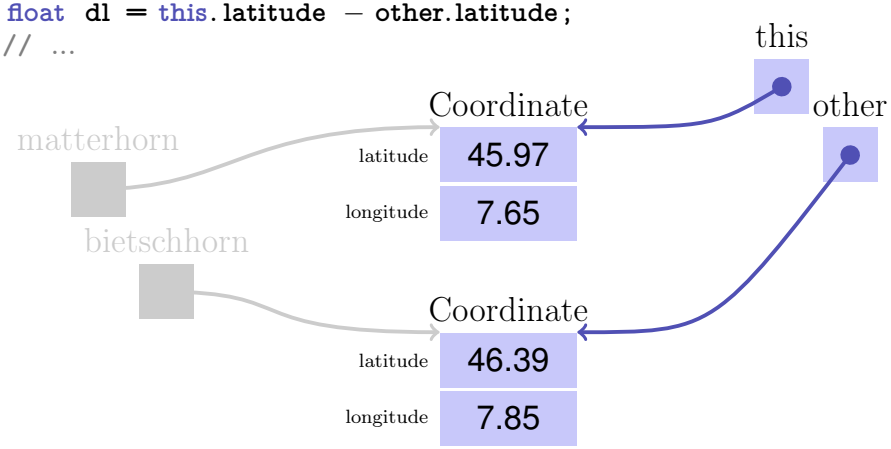

Method calls - Example setup

```
Coordinate matterhorn, bietschhorn;
// ... Instanciate and set values ...
d = matterhorn.distance(bietschhorn);
```



From the context inside the method

```
float distanz(Coordinate other){
    float dl = this.latitude - other.latitude;
    // ...
```



Keyword `this`

`this` enables to access the current object from within a method of that class.

Constructors

Creating a `Coordinate` is somewhat cumbersome:

```
Coordinate k = new Coordinate();
k.latitude = 45.97f
k.longitude = 7.65f
```

Constructors facilitate to easily set the initial values of a newly created object.

```
Coordinate k = new Coordinate(45.97f, 7.65f);
```

In general, the job of the constructor is to establish a reasonable "valid" state.

Constructors - Definition

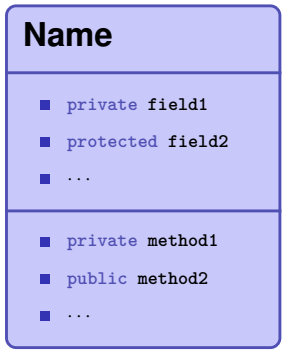
```
public class Coordinate{
    float latitude;
    float longitude;

    // Constructor for a given coordinates (as a pair of lat/long).
    Coordinate (float lat, float lon){
        this.latitude = lat;
        this.longitude = lon;
    }

    // Default constructor without parameters
    Coordinate(){}
}
```


Data Encapsulation / Information Hiding

Control, what data and what code can be *accessed* from where.

Access modifiers:

- `private`: Visible only from code within the same class
- `protected`: Visible from code in the same class or a subclass (later)
- `public`: Visible from everywhere



Example: Coordinate

```
public class Coordinate {
    public double latitude;
    public double longitude;

    public double distance(Coordinate other){...}
}
```

Problems:

- Assignment of invalid values
- Consistency checks not possible
- Implementation exposed

Coordinate: Accessor Methods

```
public class Coordinate {
    private double latitude;
    private double longitude;

    public double getLatitude(){
        return latitude;
    }

    public void setLatitude(double lat){
        assert latitude < -90 || latitude > 90;
        this.latitude = lat;
    }
    //...
```


Coordinate: Usage

```
Coordinate position = ...;
position.setLatitude(45);     //This is fine

Out.println(position.getLatitude());     //This is fine


// The following two lines are WRONG
position.setLatitude(100);     //Assertion violation at runtime
Out.print(position.latitude); //Doesn't compile. Invalid access
```


Encapsulation: Exchange implementation

With no direct access to the data, it is easy to change the implementation without making it visible "to the outside".

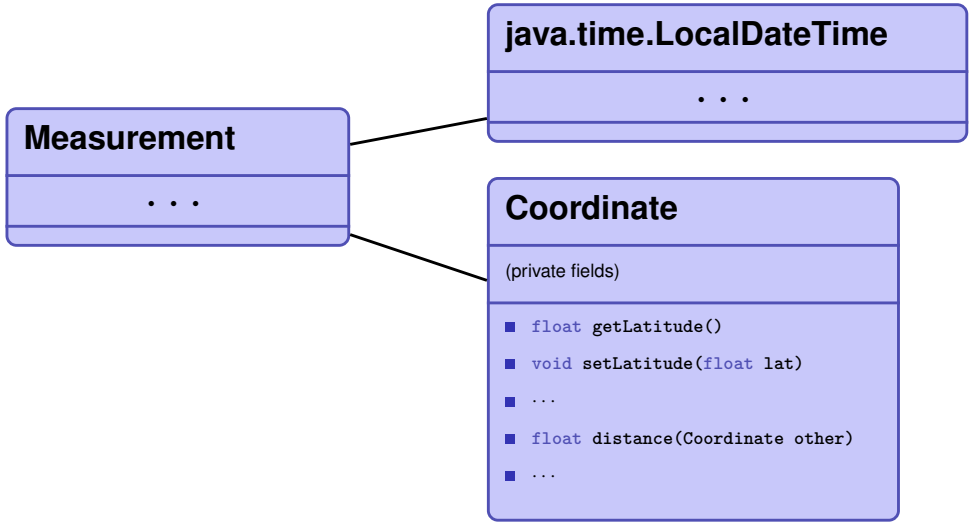
Example: Switch to Swiss Coordinate Grid

```
public class Coordinate {
    // Coordinate in WSG84
    private int latitude;
    private int longitude;

    public double getLatitude(){




        return this.latitude;
    }
```

Example: Switch to Swiss Coordinate Grid

```
public class Coordinate {
    // Coordinate in LV03 Format (Swiss coordinate grid)
    private int x;
    private int y;

    public double getLatitude(){
        double x_aux = (x - 200_000) / 1_000_000;
        double y_aux = (y - 600_000) / 1_000_000;
        double result = (16.9023892 + (3.238272 * x_aux)) \
          - (0.270978 * pow(y_aux, 2)) \ - (0.002528 * pow(x_aux, 2)) \
          - (0.0447 * pow(y_aux, 2) * x_aux) \ - (0.0140 * pow(x_aux, 3));
        return (result * 100) / 36;
    }
```


Class Design - third try



Data Encapsulation

- A complex functionality gets defined as abstract as possible semantically and made accessible trough an agreed-upon minimal *interface*
- It should not be visible for the client *how* the state is represented in data fields of the class
- The class provides functionality to the client *independently of its representation*
- This allows to enforce *invariants*

Static Fields and Methods

Declared with the keyword `static`.

- Exist only once per class
- Are accessed directly via the class name rather than objects of the class. . .
- . . . this is why it's not possible to access `this` from static methods.
- Observation: the `main` method is static!
  `public static void main(String[] args)`

Example: The `In` class

```
float f = In.readFloat()
```

Is defined in class `In`:

```
public class In {
    public static float readFloat(){
        String s = readFloatDigits();
        try {
            done = true;
            return Float.parseFloat(s);
        } catch (Exception e) {
            done = false; return 0f;
        }
    }

    // ...
```