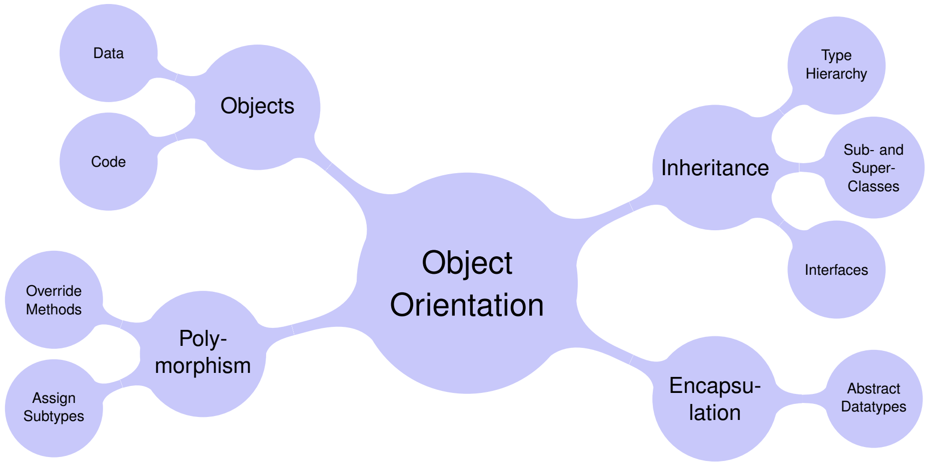
14. Java Object Orientation

Classes, Inheritance, Encapsulation

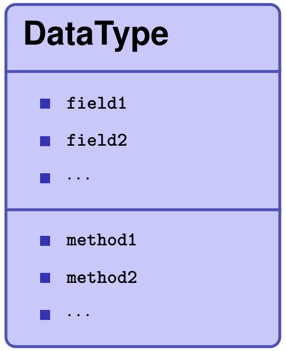
Object Orientation: Different Aspects



Already discussed: Objects

Focus on *Object* of a data type that contain

- Data (Fields) and
- Code (Methods)

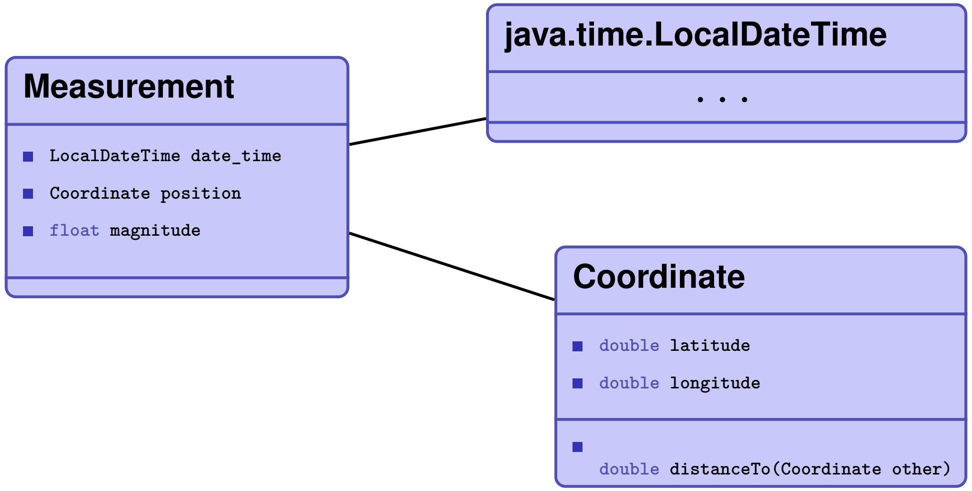


DataType

- field1
- field2
- ...

- method1
- method2
- ...

Already discussed: *Composition* of Objects



Measurement

- LocalDateTime date_time
- Coordinate position
- float magnitude

java.time.LocalDateTime

. . .

Coordinate

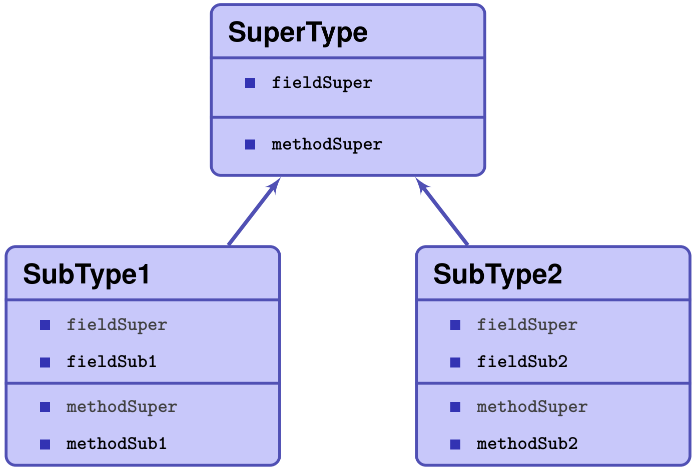
- double latitude
- double longitude

- double distanceTo(Coordinate other)

Inheritance

Data types are part of a type hierarchy

Subtypes inherit data and code from their *supertypes*.



Inheritance ≠ Composition

Composition: An object contains fields that refer to objects of a different type

Inheritance: An object of some type contains additional fields and methods that are inherited from a supertype

Correct Use for Inheritance

Important question to identify whether `DataType1` should inherit from `DataType2`:

Is `DataType1` a `DataType2`?

Example

- *Is* a “Student” a “Person” ✔
- *Is* an “Apple” a “Fruit” ✔

Correct Use for Composition

Important question to identify whether `DataType1` should contain `DataType2` as composition:

Has `DataType1` a `DataType2`?

Example

- *Has* a “Student” an “Address” ✔
- *Has* an “Apple” a “Color” ✔

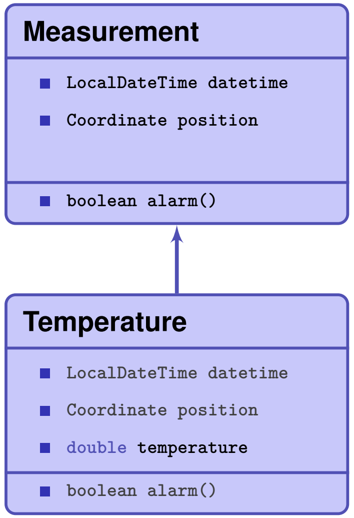
Inheritance: `extends` Keyword

```
class Measurement {
    LocalDateTime datetime;
    Coordinate position;

    boolean alarm() {...}
}

class Temperature extends Measurement {
    double temperature;
}

class Wind extends Measurement {
    double speed;
    double direction ;
}
```



Measurement

- LocalDateTime datetime
- Coordinate position

- boolean alarm()

Temperature

- LocalDateTime datetime
- Coordinate position
- double temperature

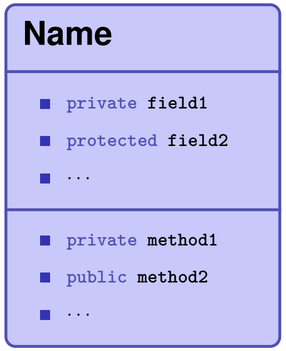
- boolean alarm()

Data Encapsulation (Repetition)

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
- `public`: Visible from everywhere



Name

- private field1
- protected field2
- ...

- private method1
- public method2
- ...

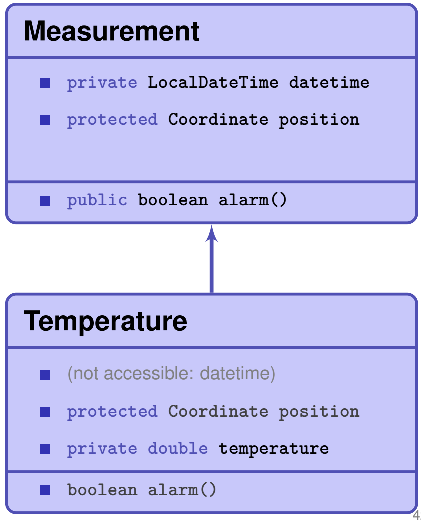
Example for `protected` Visibility

```
class Measurement {
    private LocalDateTime datetime;
    protected Coordinate position;

    public boolean alarm() {...}
}

class Temperature extends Measurement {
    private double temperature;
}

class Wind extends Measurement {
    private double speed;
    private double direction ;
}
```



Measurement

- private LocalDateTime datetime
- protected Coordinate position

- public boolean alarm()

Temperature

- (not accessible: datetime)
- protected Coordinate position
- private double temperature

- boolean alarm()

Abstract Classes

```
class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    public boolean alarm() {...}
}
```

- Class `Measurement` provides a method `alarm()`
- The method should return `true` *iff* the measurement is *alarming* ...
- *... but the implementation of the method depends on the implementation of the different subtypes ... ?!*

Abstract Classes

It doesn't make sense to create objects of type `Measurement`, it should be *abstract*.

Abstract Classes: Keyword `abstract`

```
abstract class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    abstract boolean alarm();
}

class Temperature extends Measurement {
    double temperature;

    // Implement the abstract method from the supertype
    boolean alarm(){
        return temperature > 35;
    }
}
```

Abstract Classes: Keyword `abstract`

```
abstract class Measurement {
    ...
    // returns 'true' if measurement is alarming, 'false' otherwise
    abstract boolean alarm();
}

class Wind extends Measurement {
    double speed;

    // Implement the abstract method from the supertype
    boolean alarm(){
        return speed > 80;
    }
}
```

Abstract Classes: Properties

- If at least one method is `abstract`, that is, not implemented, the whole class has to be declared `abstract`.
- Abstract classes *can't* be instanciated (`new ...`)
- Abstract classes contain data and code that is inherided by all subtypes. The differences are abstracted.

Abstract Classes: Usage

```
Temperature t = new Temperature(40);
boolean b = t.alarm();
```

⇒ In his example, the variable `b` is set to `true`.

What if we call `alarm()` *from a method defined in class* `Measurement`*?*

Abstract Classes: Dynamic Method Binding

```
abstract class Measurement {
    abstract boolean alarm();

    String alarmOutput(){
        if (this.alarm()){
            Out.println("Alarm!");
        } else {
            Out.println("Nominal");
        }
    }
}
```

Abstract Classes: Dynamic Method Binding

```
Temperature t = new Temperature(40);
t.alarmOutput();
```

⇒ Outut: `"Alarm!"`

- The object `t` of type `Temperature` inherited the method `alarmOutput`.
- In this object, the implementation from method `alarm()` in Class `Temperature` is bound to the abstract method `alarm()`.
- Thus, `alarmOutput()` will call the implementation from `Temperature`.