14. Java Object Orientation

Classes, Inheritance, Encapsulation

Already discussed: Objects

Focus on \textit{Object} of a data type that contain
- Data (Fields) and
- Code (Methods)

Already discussed: \textit{Composition} of Objects

\begin{itemize}
  \item \texttt{java.time.LocalDateTime} \texttt{...}
  \item \texttt{Coordinate} \texttt{...}
  \item \texttt{double latitude}
  \item \texttt{double longitude}
  \item \texttt{double distanceTo(Coordinate other)}
\end{itemize}

\begin{itemize}
  \item \texttt{LocalDateTime} \texttt{date\_time}
  \item \texttt{Coordinate} \texttt{position}
  \item \texttt{float magnitude}
\end{itemize}
**Inheritance**

Data types are part of a type hierarchy.

*Subtypes* inherit data and code from their *supertypes*.

**SuperType**
- fieldSuper
- methodSuper

**SubType1**
- fieldSuper
- fieldSub1
- methodSuper
- methodSub1

**SubType2**
- fieldSuper
- fieldSub2
- methodSuper
- methodSub2

---

**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**

```java
class Measurement {
    LocalDateTime datetime;
    Coordinate position;
    boolean alarm() {...}
}

class Temperature extends Measurement {
    double temperature;
}

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

**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

**Example for protected Visibility**

```java
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;
}
```

**Abstract Classes**

```java
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: 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 inherited 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");
        }
    }
}

Temperature t = new Temperature(40);
t.alarmOutput();
⇒ Output: "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.