Technische Hochschule **Rosenheim**
Technical University of Applied Sciences

Object-oriented programming
Chapter 1 – Professional software development

Prof. Dr Kai Höfig

What we’ve covered so far...

- The following topics were covered in the Programming Basics lectures:
    - Fundamental language concepts
    - Control structures
    - Object orientation
    - Classes
    - Characters and strings
    - Arrays and containers
    - Packages
    - Inheritance
    - Exceptions

Professional software development

The key elements of professional software development are

1. Modelling the problem, ideally before the start of the implementation. The modelling is based on the problem and the resulting specification.

2. Implementation of the functionality.

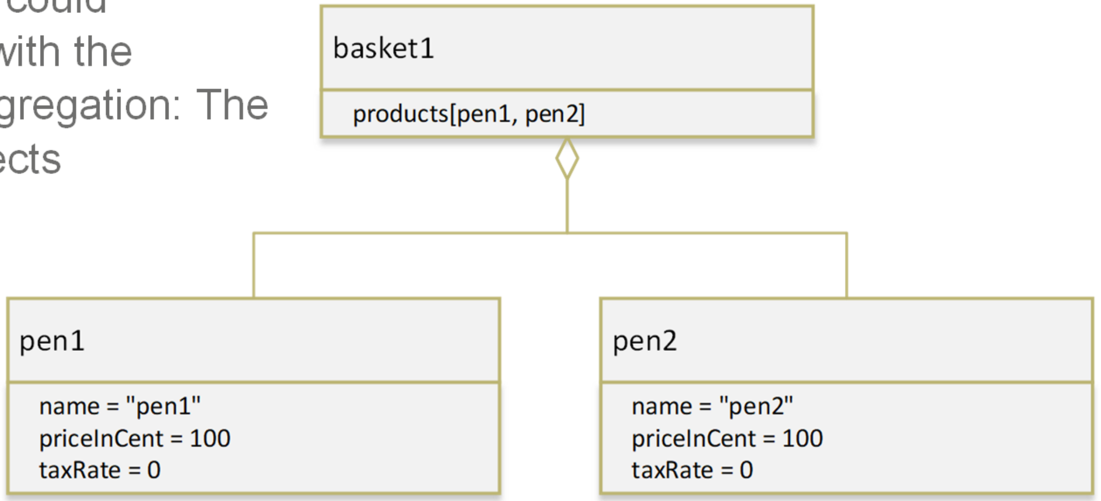
3. Implementation of tests to test the functionality.

4. Versioning.

Modelling an example with UML
Revision of the object diagram

- Let's start with a small example. For a web shop system, we initially need *Products* which have a name, price and tax rate (e.g. 19% or a reduced rate of 7%). Products are then added to a *Basket*.

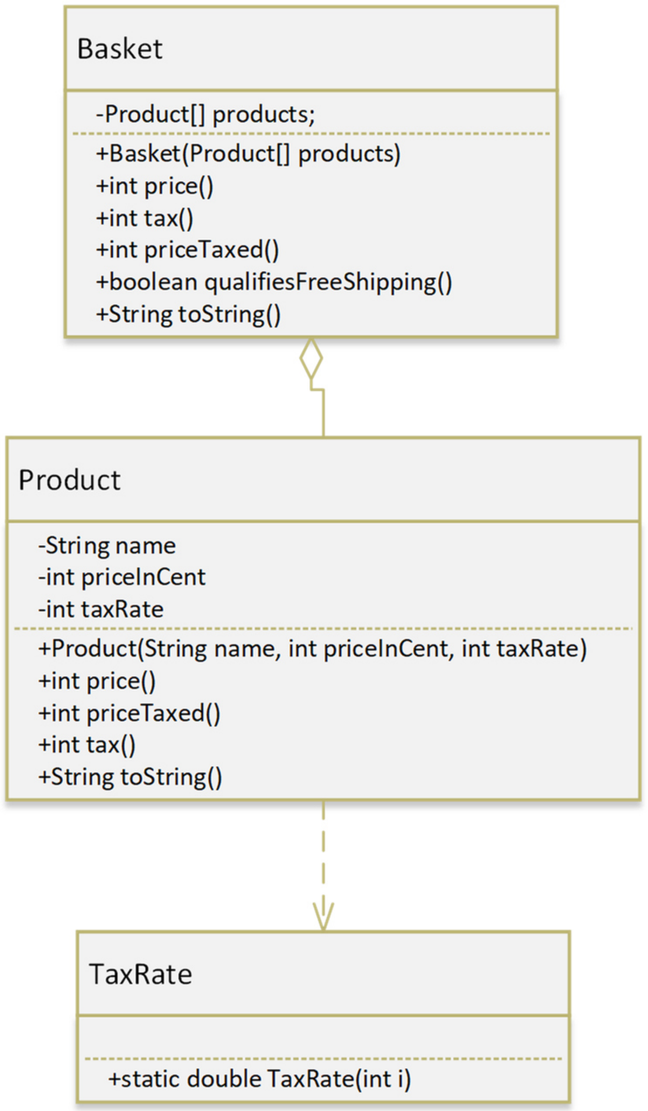
- A corresponding ***object diagram*** could look like this, whereby the arrow with the empty rhombus represents an aggregation: The object `basket1` contains the objects `pen1` and `pen2`.



- Please note: A **composition** is represented by an arrow with a filled rhombus; unlike aggregation, it indicates that an object cannot exist without its parts. In this case, the basket would consist of the pens instead of just containing them. This semantics is *common sense.*

Modelling an example with UML
Revision of the class diagram

- If we want to implement this small model world in an object-oriented language such as Java, we abstract the diagram into a *class diagram*.
- In addition to the associations, the visibility is modelled here: the - stands for **private**, i.e. not accessible from the outside, the + for **public**. We usually specify attributes as private, but methods as public. This principle is also called encapsulation.
- In addition to the two classes of Basket and Products, the TaxRate is now also modelled here. Strictly speaking, this is not an object, since only the actual TaxRate is configured here. Accordingly, it is **associated** with the Product, represented by a dashed line. The Product uses the TaxRate to calculate the price including tax via `priceTaxed()`.



Implementation in Java TaxRate

The TaxRate could now be implemented as follows:

```java
public class TaxRate {

    public static double taxRate(int i) {

        switch (i) {

            case 0: return 0.19;

            case 1: return 0.07;

            default: throw new IllegalArgumentException(i + " is not a valid tax rate");
        }
    }
}
```

Thus, the normal (0) and reduced (1) TaxRates are encoded, and for other requests an unchecked exception of the type `IllegalArgumentException` is thrown (realised by the default statement).

Implementation in Java Product

The Product could now be implemented as follows:

The visibility is set by the keywords `private` and `public`. If there is shadowing of variables, such as the constructor argument `name` and the attribute `name`, then the self-reference `this` can be used to get access to the shadowed attribute.

```java
public class Product {
  private String name;
  private int priceInCent;
  private int taxRate;

  public Product(String name, int priceInCent, int
taxRate) {
    this.name = name;
    this.priceInCent = priceInCent;
    this.taxRate = taxRate;
    }

  public int price() {
    return priceInCent;
    }

  public int priceTaxed() {
    return priceInCent + tax();
    }

  public int tax() {
    return (int) Math.round(TaxRate.taxRate(taxRate) *
priceInCent);
    }
}
```

Implementation in Java
Basket

The Basket could now be implemented as follows:

Here we see that the Products were realized as an array, whose content can be iterated with a for loop.

To be revised if necessary

- Creating classes, objects and arrays
- Conditions with if and else
- Iteration with for and while
- Checked and unchecked exceptions, try-catch-throw

```java
public class Basket {
  private Product[] products;

  public Basket(Product[] products) {
    this.products = products;
    }

  public int price() {
    int p = 0;
    for (Product w : products)
      if (w != null)
            p += w.price();
    return p;
    }

  public boolean
qualifiesFreeShipping() {
    return price() >= 300;
    }

  /* ... */
}
```

Testing with JUnit (5)

- In addition to implementing the classes and methods, professional software development requires testing for correctness, whereby testing is generally not able to provide absolute proof of correctness. There are various approaches and toolkits that can be used to simplify or automate the testing - in this course we use JUnit 5.
- During testing, the test files are *often* kept in separate directories, as they are not delivered to the customer. For example, the application code is usually stored under src/main/java, but the test code is usually stored under src/test/java. However, we will not be doing this in the lecture project, in order to maintain a better overview.
- The test driver (e.g. IntelliJ or Gradle) then performs all methods annotated with `@Test` as individual test cases.
- Simple tests that isolate a class or method (or consider a simple interaction) are known as **unit tests**. They make sure that the components do what they are supposed to do "on the small scale".
- If the software (or a significant part of it) is tested in its entirety, these are known as *integration tests*. These then ensure that the separate components (which have already been tested individually) work together correctly, so that the software delivers the desired result.

Example JUnit 5 test case

- A test in JUnit is a class with specially annotated methods, for example:

```
class TaxRateTest {
  @Test
  void testTaxRate() {
    Assertions.assertEquals(0.19, TaxRate.taxRate(0));
    Assertions.assertEquals(0.07, TaxRate.taxRate(1));

    Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
      @Override
      public void execute() throws Throwable {
        TaxRate.taxRate(-1);
            }
        });

    Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
      @Override
      public void execute() throws Throwable {
        TaxRate.taxRate(2);
            }
        });
    }
}
```

- JUnit provides helper methods that make testing easier. Because most tests rely on the fact that software produces specific output for specific input, there is a list of helper methods in the `Assertions` class to check expected behaviour.
- Thus, `Assertions.assertEquals(0.19, TaxRate.taxRate(0))` checks that the return value of `TaxRate.taxRate(0)` is equal to 0.19.

Assertions

- The following slightly cumbersome code snippet checks whether an exception is thrown for arguments other than 0 or 1:

```
Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
  @Override
  public void execute() throws Throwable {
    TaxRate.taxRate(2);
    }
});
```

- By the way, since Java 8, this can be written in short form as a *lambda expression*:

```
Assertions.assertThrows(IllegalArgumentException.class,
        () -> TaxRate.taxRate(-2));
```

Overview of the Assertions class

https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

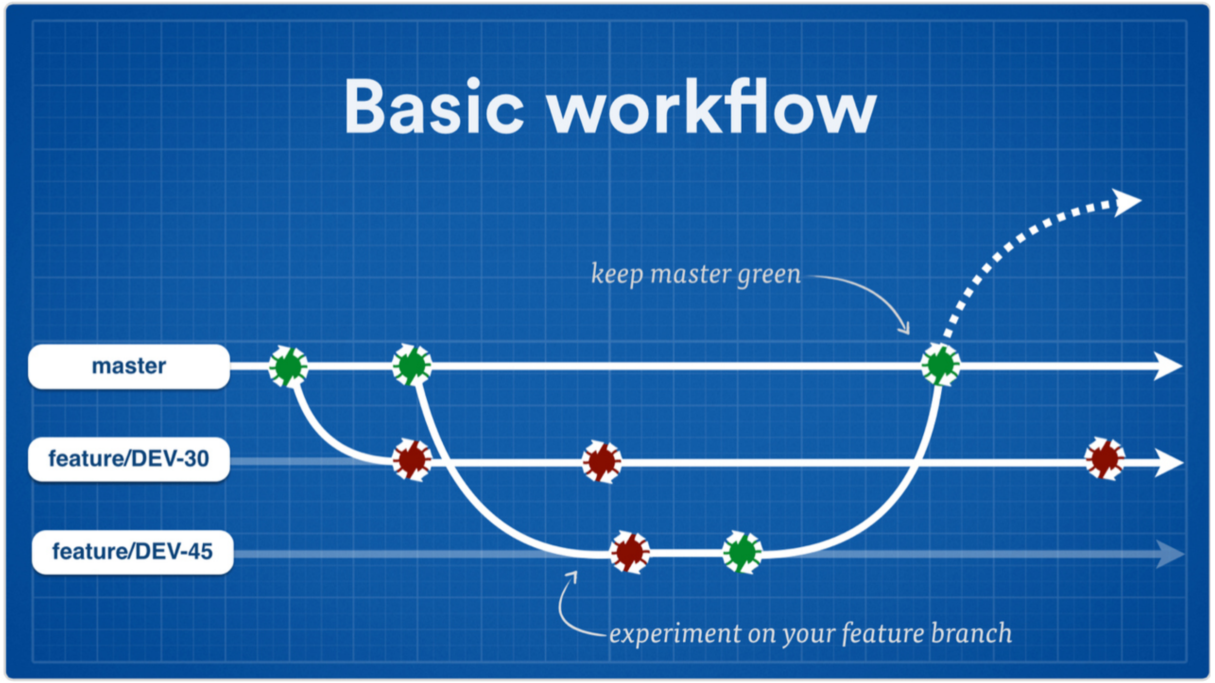
Brief introduction to Java lambda expressions

https://www.youtube.com/watch?v=GxZWMgpMuLs

Versioning with Git

- Often, multiple developers work together on a project, and it is wise to create backup points (snapshots) at regular intervals for backup and documentation purposes.

- The versioning software Git helps with this. In simple terms, there should be a main inventory of the source code (*master*), and new features should then each be implemented in separate *branches*. If a feature is finished in a branch, it is then included in the master by a *merge*:



Instructions for using Git are available at
http://rogerdudler.github.io/git-guide/
https://learngitbranching.js.org/?locale=de_DE

Instructions for integrating Git (not GitHub!) in IntelliJ are available at
https://www.jetbrains.com/help/idea/using-git-integration.html