Java training

Java OOP overview - entities

Session overview

1. Object-oriented programming (OOP) overview
2. Java OOP entities and concepts
   a. Classes
   b. Objects
   c. Constructors
   d. Fields
   e. Methods
   f. `static`, `final` & `this` keywords

Object-Oriented Programming (OOP)

- Programming model based on *classes* and *objects*
- Based on **two main types of entities**:
  - *Data* - objects which *contain* data (as fields / attributes)
  - *Methods (/ functions)* - constructs used to *process* data
- The two types of entities interact to solve a business domain requirement
- Main structural unit for holding them - *classes*

Classes & objects

- **Class** - template for creating objects (of that class type)
- Classes can have:
  - Fields / properties
  - Methods
  - Behavior

  further presented
- Creating objects from a class - some properties can be automatically initialized (further presented)
- The usage of classes & objects → object-oriented programming (OOP)

Java objects and their properties

- Classes can have:
  - *Fields* / *properties* - used to hold information / data
  - *Methods* - contain the logic for processing / using the class fields
- **Constructors** - methods which have the same name as the class
  - Used for initializing the class fields / properties
  - Cannot have a return type
  - Every class:
    - Has a default constructor - has no parameters
    - Can have any number of constructors

Product class anatomy

```
public class Product {
    private String name;

    public Product() {}

    public Product(String name) {}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

- `public class Product {` → Class name
- `private String name;` → Field / variable
- `public Product() {}` → Default constructor
- `public Product(String name) {}` → Constructor
- `public String getName() { … }` → Getter
- `public void setName(String name) { … }` → Setter

Methods

Hands-on

Examples

Instantiating the `Product` class → creating objects from it:

```
Product product = new Product();   // the default constructor is used
product.setName("Tablet");         // using the setter, to set the name
System.out.println("It's a " + product.getName()); // using the getter
```

Hands-on:

- Create and use several (1-2) products of your choice
- Create a class named `ProductService` + a main method, to handle the products

Fields and methods

- **Fields** (properties, members, variables) - used to *hold* the data
- **Methods** (functions) - used to *process* the data (held by fields)
- Depending on their purpose, classes are usually used as:
  - **Data *holding* classes** - aka 'POJO (Plain Old Java Object)' or 'bean' classes
    - They have several fields + getters & setters for them
    - Used to carry data between processing classes
  - **Data *processing* classes** (service classes)
    - Contain methods → hold the business logic used to process POJO classes
    - May create / aggregate POJOs

'Separation of Concerns' (S from the SOLID principle)

Constructors - main rules

- Default constructor (the class name) - always available, even if not created
- If a class has 1+ constructors with parameters and no explicit constructor - can be *only* instantiated using the declared constructors
- Private constructors can be also used → the class cannot be instantiated (further discussed)

Methods - syntax template

```
[access-specifier] [modifiers] returnType methodName([parameter 1], ... [parameter n]) {
    [method body]
}
```

Return type

(optional)
Package access specifier
(further presented)

(optional)
Various modifiers
(further presented)

Method name
(camelCase naming)

(optional)
Method parameters

Methods - examples

- `public static void main (String[] args) {...}`
  - The start method of every Java program
  - Access specifiers: `public` & `static` (further presented)
  - Return type: `void`
  - Parameter: an array of `Strings`
- `private void processProduct(Product product) {...}`
  - Access specifiers: `private` (further presented)
  - Return type: `void`
  - Parameter: an object of type `Product`
- `protected Product createProduct(int id, String name) {...}`

Hands-on

`static` entities

- **'`static`'** (fields & methods) - belong to *all* the instances from a class, not to a single instance
- Examples:

```
class Product {
    private static final String TABLE_NAME = "product";
}
```

field

→ a `String` which holds the table where the products are stored

```
public static Product createProduct(int id, String name) {
    return new Product(id, name);
}
```

method

→ belongs to all instances, does not need an instance to be used

Using `static` entities

- **Fields:**
  - `static` fields can't be accessed from non-`static` methods → compilation error
  - They occupy a single memory space, for all the instances
- **Methods:**
  - Invoked directly on the class, not on an object (class instance):

Correct usage

```
Product product = Product.createProduct(10, “Tablet”);
```

```
Product another = new Product();
another.createProduct(11, “iPad”);
```

Incorrect usage

→ leads to an IDE warning

Hands-on

`static` initialization block

- Classes can have `static` initialization blocks → used to:
  - Initialize `static` variables when the class is initialized *for the 1st time*
  - Ensure the initialization occurs just a single time (the 1st class initialization)
- Example:

```
class Product {
    static {
        System.out.println("Initializing the class...");
    }
}
```

Hands-on

‘final’ modifier

- Modifier which makes an entity `final`
- Applicable in several contexts:
  - Classes - the class cannot be extended
  - Fields / attributes - the field can only be assigned once
    - It’s internal value(s) can be further changed (further presented)
  - Methods - the method cannot be overriden (by extending classes)
- Used in conjunction with the `static` modifier → defining *constants*:

```
private static final LocalDate TODAY = LocalDate.now();
```

All caps → naming convention for constants

Hands-on

Rules for ‘`final`’ variables / fields

- After assigning a `final` variable - will always contain the same value:

```
private static final int STORE_ID = 121;
```

- If a final variable holds a *reference* to an object:
  - The variable will *always refer to the same object* (→ *non-transitivity*)
  - The *state of the object may be changed* (by operations on the object)

```
private final List<String> list = new ArrayList<>(5);
list.add(“A smile”);            // OK
list = new LinkedList<>(5);     // compiler error → cannot re-assign
```

  - Also applies to arrays of objects

Hands-on

‘Blank `final`’ variables

- ‘**Blank `final`**’ variable - a `final` variable declared un-initialized:

```
private [static] final String blankFinal;
```

- Can be initialized in two ways:
  - 1. If it is `static` - in a `static` block:

```
static {
    blankFinal = “A blank final variable”;
}
```

  - 2. If it is non-`static` - in the constructor(s):

```
this.blankFinal = “A blank final variable”;
```

Hands-on

‘this’ keyword

- ‘this’ - reference to the current object
- Usefulness - avoid ambiguity when accessing fields & methods from a class
- Example:

```
public class Tablet {
    private String name;

    public Tablet (String name) {
        this.name = name;
    }
}
```

→ this - used to avoid the ambiguity between the class and the constructor param → pointer to the current object

Hands-on

‘this’ keyword - rules

- As it points to an object - cannot be used in a `static` context:
  - From `static` methods
  - From `static` blocks
  - To initialize `static` variables
- Can be used to access:
  - Constructors `this(10, “Tablet”);`
  - Fields `this.name = name;`
  - Methods `this.processProduct(product);`

Pass by reference or pass by value?

- Java is a **pass-by-value** programming language
- When we are passing a reference to an object, we pass *a pointer to the address of that object*

More info - here & here

Q & A session

1. You ask, I answer
2. I ask, you answer