Java training

`Abstract` classes and interfaces

Session overview

- Abstract classes and methods
- Interfaces

Abstract classes and methods

- **Abstract class** - class which is declared `abstract`
  - It cannot be instantiated → it needs to be extended, first
  - May or may not have `abstract` methods:
    - Methods declared `abstract` cannot have a method body
    - If a class has at least one `abstract` method - it must be declared `abstract`
- Example:

```
abstract class AbstractProduct {   → abstract class definition
    abstract String getName();     → abstract method definition

    int getId() { return this.id; }
}
```

Main usages

Defining an hierarchy of:

- Classes where each subtype *must* implement one or several methods
- Classes which have a common parent class, which cannot be instantiated
  - Usually called *marker class*
- Multiple `abstract` classes, to use them in a common way, through their parent class

Rules & good practices

- When a class extends an `abstract` class, it must either:
  - Implement it's `abstract` methods
  - Be also defined `abstract`
- Abstract classes can contain both `abstract` and non-`abstract` methods
  - Extending concrete (non-`abstract`) classes must only implement the `abstract` methods
- Classes which are declared `abstract` should be prefixed by the name Abstract

Abstract class hierarchy example

```
abstract class AbstractProduct {
    abstract String getName();
}

class Tablet extends AbstractProduct {
    private String name;
    public String getName {
        return this.name;
    }
}
```

→ a concrete implementation

+ setting via constructor

+ Hands-on

→ the `Product` class as an abstract class example → a `Product` is abstract (by itself)

Interfaces

- **Interface** - a structure / entity type (similar to a class) that can contain only:
  - *Method signatures*
  - Constants
  - Default methods (Java 8+)
  - Static methods
  - Nested types
- Main usability - *defining the contract* of a functionality → defining *what* it does, not *how* it is implemented
  - Example: retrieving the information about an User entity → the interface returns an User, but it does not specify *how* (/ from where) to retrieve him / her

Rules & good practices

- An `interface` needs to have at least one implementation, to be used
  - Implementing an interface → using the `implements` keyword
- All * the methods defined in an interface are implicitly `abstract` → no need to be specified `abstract` (redundant)
- Good practice (whenever possible) - the name of an interface should represent an adverb → some JDK interfaces examples:
  - `Serializable`
  - `Runnable`
  - `Callable, Closeable`

* - pre Java 8

Interface hierarchy sample

```
interface UserRepository {      → an user retrieving repository
    public User getUser(String email);      → retrieving based on email
}

class DatabaseRepository implements UserRepository {
    public User getUser(String email) {
        // code to retrieve the user from a database table
    }
}
```

+ Hands-on

The `UserRepository` **defines the contract (*what*)** - an user will be retrieved by email
It's implementations will implement ***how & from where*** the user will be retrieved

Q & A session

1. You ask, I answer
2. I ask, you answer
    a. What is the usefulness of the `toString()` method?
    b. When should we use `equals()`? Do we need to override it?
    c. Why is `hashCode()` useful?