Java – Basics of object-oriented programming

Java is a programming language for object-oriented programming. This means to solve technical problems by the use of classes. First one will model the entities in scope as Java classes, before one will solve the specific problems using these classes .

A class represents an entity of the area of concern. Each class has a unique name, a constructor (named as the class itself), specific attributes and specific functionality implemented as so-called methods.

Each Java class will be saved in a source code file ClassName.java.
Classes are grouped into packages according to technical aspects.

The mini code examples shown here are used to model the collection and refund of beverage packaging. The purpose is to illustrate key Java programming features based on minimalistic working examples.

Beverage packaging	
Flask:	general packaging such as Tetra Pack, screw-top jar, plastic bottle, metal-can;
Bottle:	glass bottle;
Can:	refundable aluminum can;

Collective packaging or container	
FlaskPack:	general collective beverage packaging of defined capacity;
QuadPack:	4-pcs-container for empty flasks
SixPack:	6-pcs-container for bottles

1 Basic design of a Java class

Name of the package, the class is related to

Class definition incl. definition of internal variables (attributes)

Constructor: named equally to the class; creates instances of the class; no return data type information required; used to initialize internal variables

Method (activity/function) of the class including input parameters.
The data type of the returned value must be stated right before the method name.

The keyword "this" is used within a class implementation to highlight explicitly that a certain class internal variable or method is meant.

```java
package org.htwd;

public class Flask {
    private double refund;

    public Flask(double refund) {
        if (checkRefund(refund) == true)
            this.refund = refund;
    }

    private boolean checkRefund(double refund) {
        if (refund < 0.0 || refund > 5.00)
            return false;
        return true;
    }

    public double getRefund() {
        return this.refund;
    }
}
```

2 Modifier

By the use of the access modifiers private, protected and public one can define the access to attributes and methods.

private:	grants access within a class, but not in sub-classes
protected:	grants access within a package and in sub-classes
public:	grants general access

static as a modifier states an attribute or method is directly associated to the class. So it can be used directly referenced by the class name and not by the name of an object variable.

final as a modifier states an attribute value can not be changed after it was assigned an initial value. Methods labeled final are not allowed to be overwritten. Classes labeled final are not allowed to get sub-classed.

3 Extension / Inheritance

Extension (inheritance) is a programming technique to design a specialized sub-class out of a base class. The sub-class takes all attributes and methods of the base class but it may define additional attributes and methods. It is possible to re-implement (override) base class methods with a different coding.

extends: The Bottle class is a sub-class of Flask.

From inside the Bottle-constructor the Flask-constructor gets called.

@Override: The getRefund-method of the above class will be overwritten for the Bottle-class. This means it gets implemented in a different way to realize a class-specific functionality.

The keyword **super** explicitly highlights variables or methods of the direct above class. This is the class named after “extends”.

```java
package org.htwd;

public class Bottle extends Flask  {
    public final static String MATERIAL = "glass";
    private boolean returned = false;

    public Bottle() {
        super(0.25);
    }

    @Override
    public double getRefund() {
        if (this.returned)
            return 0.0;
        this.returned = true;
        return super.getRefund();
    }

}
```

4 Interface

An interface defines a set of method signatures, a class needs to implement. The interface itself will not implement any method.

interface: Definition of the interface Refundable. It just defines the method returnIt.

```java
package org.htwd;

public interface Refundable {
    double returnIt();
}
```

implements: The class Can implements the interface Refundable. Furthermore this class is derived from the Flask class.

@Override: The implementation of the returnIt-method ensures, the interface Refundable is completely coded.

```java
package org.htwd;

public class Can extends Flask
                 implements Refundable {

    private boolean returned = false;

    public Can(double refund) {
        super(0.25);
    }

    @Override
    public double returnIt() {
        if (this.returned)
            return 0.0;
        this.returned = true;
        return getRefund();
    }
}
```

5 Using objects

Classes are made to be used and to interact with other classes. Therefore one needs to create an object/instance of the class with help of the new-statement. The result of the new-statement is a new object of the class that will be assigned to a variable. This object-variable will be used to call methods of the class in order to make use of the classes functionality.

import: Import the ArrayList class out of the package java.util to be used here.

new: Create an object of class ArrayList to store Bootle objects inside it.

Create a Bottle object and save it directly into the ArrayList variable pack by calling the add-method.

This method returns a Bottle object by removing it from the pack ArrayList.

```java
package org.htwd;

import java.util.ArrayList;

public class SixPack {
    private final int SIX = 6;
    ArrayList<Bottle> pack = new ArrayList<Bottle>();

    public SixPack()
    {   int k = 0;
        while (k<this.SIX) {
            this.pack.add(new Bottle());
            k++;
        }
    }

    public Bottle popBottle() {
        Bottle b = null;
        if (this.pack.size() > 0)
            b = this.pack.remove(0);
        return b;
    }
}
```

6 Abstract classes

An abstract class defines basic properties and functionality the deriving class needs to implement. Besides this an abstract class can be used to define abstract methods to be implemented by the sub-classes. It is not allowed to create objects of abstract classes.
An abstract class is more or less a conceptual bases for deriving classes.

abstract: The FlaskPack class is an abstract class containing a constructor, an attribute variable and a method.

Furthermore it defines the methods takeOne and putOne as abstract methods, to be implemented by sub-classes.

```java
package org.htwd;

public abstract class FlaskPack {
    private final int LIMIT;

    public FlaskPack(int limit) {
        this.LIMIT = limit;
    }

    public int getLimit() {
        return this.LIMIT;
    }

    abstract public Flask takeOne();
    abstract public boolean putOne(Flask flask);
}
```

The QuadPack class is a sub-class of the abstract class FlaskPack. It implements the required methods takeOne and putOne.
It declares the attribute pack as a vector to store Flask objects.
Furthermore it provides the count-method.

The main-method demonstrates the use of QuadPack as box of 4 drink cans.

```java
package org.htwd;

import java.util.Vector;

public class QuadPack extends FlaskPack {

    private Vector<Flask> pack = null;

    public QuadPack(int limit) {
        super(limit);
        this.pack = new Vector<Flask>(limit);
    }

    @Override
    public Flask takeOne() {
        if (this.pack.size() > 0) {
            return this.pack.remove(0);
        }
        return null;
    }

    @Override
    public boolean putOne(Flask flask) {
        if (this.pack.size() < getLimit()) {
            this.pack.add(flask);
            return true;
        }
        return false;
    }

    public int count() {
        return this.pack.size();
    }

    public static void main(String[] args) {
        QuadPack canPack = new QuadPack(4);
        canPack.putOne(new Can(0.5));
        canPack.putOne(new Can(0.5));
        int l = canPack.getLimit();
        int c = canPack.count();
        System.out.println(l+" "+c);
        Can can = (Can) canPack.takeOne();
        double amount = can.returnIt();
        c = canPack.count();
        System.out.println(can+": "+amount+" EUR");
        System.out.println(l+" "+c);
    }
}
```

7 Exception handling

Whenever in Java a risky operation gets performed one needs to secure it with help of try-except-statements. This will prevent a program crash in case of an exception. For so-called Checked-Exceptions the Java-compiler requires a mandatory try-except-handling. IO-errors and SQL-exceptions are examples for such Checked-Exceptions. Alternatively to the try-catch-statements one can pass the error to the calling method with help of the throws-keyword.

throws: The constructor may throw an IllegalArgumentException.

throw: Create and throw an IllegalArgumentException.

try: In this part of the code an IOException can occur due to the file operations.

catch: The variable e catches the IOException that can occur in the try-section. The catch-section shows the error and prevents the program from crashing. The program continues after the catch-section even in the case of error.

```java
package org.htwd;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class RefundLogWriter {

    private final String FILENAME ;

    public RefundLogWriter(String filename)
            throws IllegalArgumentException {
        if (filename == null)
            throw new IllegalArgumentException("Parameter
                            'filename' cannot be null");
        this.FILENAME = filename;
    }

    public void createNewLogFile() throws IOException {
      new File(this.FILENAME);
    }

    public void writeLog(String msg) {
        try {
            FileWriter myWriter =
                 new FileWriter(this.FILENAME, true);
            myWriter.write(msg);
            myWriter.close();
        } catch (IOException e) {
            System.out.println("Error occurred:" +
                                  e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Different exceptions that can occur in the try-section can be handled by different catch-sections.

Java method annotations like @Override generate meta-data for the compiler to generate notes or warnings. The annotation use is optional.

Hochschule für Technik und Wirtschaft Dresden
University of Applied Sciences