**Object Oriented Development using Java**

OOD Week 1 – Module 7

Dependencies

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this walkthrough cover?

This walkthrough will introduce and discuss dependencies, and how they relate to code.

# How long will the walkthrough take to complete?

Around 1-2 hours

# What should you have already completed?

You should have completed the slides on UML, and classes

# What do you need?

In order to complete this walkthrough you will need:

* Java Development Kit 1.8 or above
* Eclipse IDE Kepler or above

# What does this walkthrough cover?

* What dependencies are
* The four types of dependencies – Association , Dependency , Aggregation, and Composition
* UML and code examples of the four types

# What is a dependency?

No one class is an island. Classes depend upon other classes, even for the most basic of functionality. While we do not have to list the dependencies on Java or third party classes, we need to list any and all classes made in house that other in-house classes rely upon. The term *dependency* is meant to describe the relationship between two classes, and how do they use one another to accomplish a task.

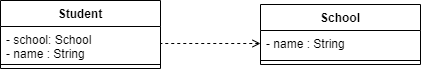
The word ‘dependency’ is both a term used in class diagrams as an umbrella term to describe a category of terms, as well as a term itself. We will be discussing the four dependencies, as well as show class diagrams and code to explain their purposes.

# The Four Dependencies

The four dependencies, from weakest to strongest connection, are: Association, Dependency, Aggregation, and Composition. Association is when the variable is stored as a value in another class as an instance variable. Dependency is for when a class is used within a method as a parameter or as a local variable. Aggregation is for a class that contains a collection of other classes, but these two classes can exist independently. Composition has the ‘strongest’ link between the two classes, insomuch the fact that one class cannot exist without the other class. If there is a situation in the creation of a class diagram wherein two classes depend on one another in multiple ways, use the stronger connection.

## Association

Association is about having one class know about another via an instance variable. In Class Diagrams, it is represented via an open arrowhead, and a dotted arrow line.



One could argue that the school’s relationship with student here could be an association if the school is used in getters/setters in the Student class. However, since we do not track getters/setters in a class diagram, those interactions are ignored as part of the class diagram.

To see an example of the association, look at the code below:

**public** **class** Student {

**private** School school;

**private** String name;

//getters, setters, and other code follows

}

**public** **class** School {

**private** String name;

//getters, setters, and other code follows

}

As you can see, the Student class has an instance of the School class as a variable. The student class can store information about the school.

## Dependency

Dependency is about having one class know about another via its usage in a method, either as a parameter or a local variable. In Class Diagrams, it is represented via an open arrowhead, and a solid arrow line.



The current Student class code looks like this:  
  
**public** **class** Student {

**private** School school;

**private** String name;

**public** **void** attendCourse(Course c) {

System.***out***.println(name + " is attending " + c.getCourseName());

}

}

And the Course code is :  
  
**public** **class** Course {

**private** String courseName;

**public** String getCourseName() {

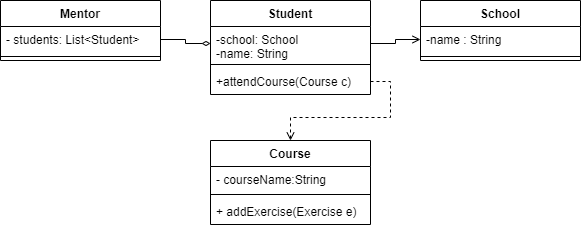
**return** courseName;

}

}

## Aggregation

Aggregation is about having one contain a collection of another class as an instance variable, but the two classes are independent of one another. In class diagrams, it is represented by a white diamond.

****

Here is the Course code, wherein it contains an aggregation relationship with the Student class.  
  
**public** **class** Mentor {

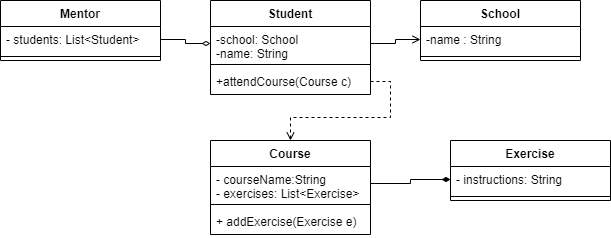
**private** Course course;

**private** List<Student> students;

}

## Composition

Composition is about having one class contain a collection of another class an instance variable, but the inner class cannot exist without the outer class. In class diagrams, it is represented by a black diamond.

****

Here is the Course code, which contains the Exercise code:  
  
import java.util.ArrayList;

import java.util.List;

public class Course {

private String courseName;

private List<Exercise> exercises;

public Course() {

setExercises(new ArrayList<Exercise>());

}

public List<Exercise> getExercises() {

return exercises;

}

public void setExercises(List<Exercise> exercises) {

this.exercises = exercises;

}

public void addExercise(Exercise exercise) {

exercises.add(exercise);

}

public class Exercise {

private String nameOfExercise;

public Exercise(String nameOfExercise) {

this.nameOfExercise = nameOfExercise;

}

public String getNameOfExercise() {

return nameOfExercise;

}

}

public String getCourseName() {

return courseName;

}

}

## Side note: Inner classes

What is a class doing inside of another class? This is called an *inner class*. It is a class that can only be accessed through another class. In this case, the Exercise class can only be directly accessed by the Course class. To initialize a new instance of Exercise, we need to go through the Course class itself, as follows:

**import** java.util.ArrayList;

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Course maths = **new** Course();

maths.setExercises(**new** ArrayList<Course.Exercise>());

Course.Exercise exercise = maths.**new** Course.Exercise("Do the even numbered problems");

maths.addExercise(exercise);

}

}

In order to access the Exercise Class, the code needs to call it through the Course class. You can use import Course.Exercise; as well.

# Conclusion

The four dependencies are key to understand the relationships between classes that do not inherit/implement one another. Understanding and recognizing the various arrows and what they mean will mean you can create and understand complex class diagrams much more easily, leading to better code structure.