**Object Oriented Development using Java**

OOD Week 1 – Module 6

Classes

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 discuss methods

# How long will the walkthrough take to complete?

2 hours

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above
* Any extra software should go here (Git, SQL, etc.)

# What does this walkthrough cover?

* Classes
* Creating an instance of a class
* Member variables
* Access Modifiers
* Constructors

# Classes

A class is like a template that can be used many times.

For instance, as a car manufacturer reuses the same template for a car, a developer can use a class to create many instances from that template.

You can create many classes of your own, for example:

* A Calculator
* An Employee
* A Bank Account

A basic class is defined as follows:

**class** Calculator

{

}

A class contains the following sections. For example:

**class** Employee

{

// Members (or member variables)

// Constructors (to construct the object)

// Methods (functionality)

}

Members Section

* The members define what data an object stores.
* For an Employee, this might include an employeeId, a name and salary.
* Only include members when the member will need to exist for the life of the object.

Constructor Section

* The constructors define the data required when creating an instance.

Method Section

* The methods provide the functionality
* When creating a class, decide on the name you want it to have(this depends on what its purpose is). Here we are storing Employee details.
* Then decide on the data and methods you want it to contain.
* The process of deciding on what data or methods it will contain is known as abstraction.

When you create an object, you are creating an instance of a class.

An object has the following attributes associated with it:

* Identity - The name of the object you are creating.
* State - The variables, and their values, inside the object
* Behaviour - The methods available to the object.

To create an object the new keyword is used. This calls a constructor of the class.

**public** **class** Runner {

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

{

Employee employee = **new** Employee();

}

}

# Creating an instance of a class

To create an instance of our Employee class do the following.

This line of code consists two parts:

Employee employee This is known as an object reference

= new Employee(); This code calls a constructor in the Employee class

to create the Employee object

In memory, the Employee object is located at the memory address stored within the reference variable called employee.

# Member variables

Declaring member variables is very similar to declaring any variable.

Any primitive or class type can be used to declare members.

Any data for an object can be used as a member variable:

* EmployeeId
* Name
* Salary

A salary increase would not need to be stored permanently but the salary would.

For example, an Employee class would store the following:

**class** Employee

{

**int** employeeId;

String name;

**double** salary;

}

We could use the Employee class in the following program:

**public** **class** Runner {

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

{

Employee employee = **new** Employee();

employee.employeeId = 123;

employee.name = "John";

employee.salary = 12345;

}

}

Note: The previous example is not good practice because all classes (not just the Employee class) have access to the members.

For good practice the members should be declared private:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

}

Now, code which uses the Employee class will not be able to access the members directly.

The solution is to use special methods called getters and setters.

The get methods are written to allow access to a member’s value. An example of this is shown below:

**public** **int** getEmployeeId(){

**return** employeeId;

}

The set methods are written to allow the changing of a member’s value. An example of this is shown below:

**public** **void** setEmployeeId(**int** employeeId){

**this**.employeeId = employeeId;

}

The code this.employeeId refers to the member variable in the class.

So we can add these methods to the class as follows:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

**public** **int** getEmployeeId(){

**return** employeeId;

}

**public** **void** setEmployeeId(**int** employeeId){

**this**.employeeId = employeeId;

}

}

We can then use them as shown to get or set the employeeId:

**public** **class** Runner {

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

{

Employee employee = **new** Employee();

employee.setEmployeeId(123);

// we call the setEmployee method

}

}

We can repeat this technique with the name and salary.

In the earlier example we encountered the default modifier:

**class** Employee

{

**int** employeeId;

String name;

**double** salary;

}

# Access Modifiers

Each of the members, employeeId, name and salary were all default access even though the default modifier was not present.

There are four access modifiers:

* A variable which is private is only accessible in the same class
* A variable which is public is accessible everywhere
* The other modifiers are:
  + protected
  + "default" or "package" access modifier. If a variable is package access the variable would be accessible anywhere in the same package as where the Employee class was defined.

This means that if several classes were defined on one package then ALL those classes could access the package access variable.

This means those classes could change the value of the variable, potentially leading to errors when the code is running.

For this reason private and public are the most widely used.

# Class Diagrams

Class diagrams are widely used to describe a class.

The main benefit is that you can use a diagram to describe how it works without writing a line of code. An example of a class diagram is shown below.

There are three sections:

* The class name
* The members with their data types
* The methods and their return types

|  |
| --- |
| Employee |
| - employeeId: int  - name: String  - salary: double |
| # otherVariable: double  + getEmployeeId(): int  + setEmployeeId(employeeId: int): void |

In the example:

* employeeId is private ( – )
* The methods are public ( + )
* The value called otherVariable is protected ( # )
* Package access variables would be indicated with a ~ symbol

# Constructors

Classes include constructors to enable “initialisation code” to be executed when an instance of a class is created. In the example earlier, we stored three member variables:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

}

We can use a constructor to set initial values for these variables. The simplest form of a constructor is shown below. It is also known as the default constructor because it has no arguments:

Employee(){

}

The code for the Employee class then becomes:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

Employee(){

}

}

Two rules:

* A constructor needs to have the same name as the class. It is case sensitive.
* A constructor must not have a return type.

The default constructor is the most basic one. One problem with the constructor above is that it does nothing. So let's change it:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

Employee(**int** employeeId, String name, **double** salary){

**this**.employeeId=employeeId;

**this**.name=name;

**this**.salary=salary;

}

}

Now the constructor passes in three arguments: an employeeId, name and a salary. To use this constructor you would write:

Employee employee = **new** Employee(1, "John", 12345.0);

It is not compulsory to include the default constructor but the default constructor is automatically created in a class if no other constructor is provided.

Constructors can be overloaded. This means that another constructor can be defined but a different signature (list or arguments) must be used.

It may be necessary to have a variety of ways to create an instance of a class.

The code below allows us to create Employees in different ways.

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

Employee(**int** employeeId, String name, **double** salary){

**this**.employeeId=employeeId;

**this**.name=name;

**this**.salary=salary;

}

Employee(**int** employeeId, String name){

**this**.employeeId=employeeId;

**this**.name=name;

}

}

Member variables can also be declared final. This means their value cannot be changed.

For example:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

**private** **final** **int** numberOfPayDays = 52;

}

One problem with this code is that numberOfPayDays is set using a hard-coded value. In the following example we can set it in the constructor:

**class** Employee

{

**private** **int** employeeId;

**private** String name;

**private** **double** salary;

**private** **final** **int** numberOfPayDays;

Employee(**int** employeeId, String name, **double** salary,

**int** numberOfPayDays){

**this**.numberOfPayDays = numberOfPayDays;

}

}

One problem with the code above is that it is not easy to identify that numberOfPayDays is a constant. Therefore it is a best practice to use capital letters for this type of declaration:

**private** **final** **int** NUMBER\_OF\_PAY\_DAYS;