Object Oriented Programming (part 1)

Agenda

- Object oriented programming fundamentals
- Inheritance
- Object oriented principles

Object oriented programming fundamentals

Overview

- We covered so far the very basics of object oriented programming in Java
- We defined what are Java classes and how to create them
- They are the fundamental unit of object oriented programming in Java
- In this session we will cover object oriented programming in Java in depth …

Basic terminology

- These are the fundamental concepts related to object oriented programming in Java:
  - **classes**: models a real world entity
  - **objects**: instances of a class
  - **packages**: namespaces for organizing classes, interfaces, enums and annotations
  - **inheritance**: a mechanism to link classes hierarchically and thus structure a software system
  - **interface**: a logical view of the operations an implementing class provides

Classes

- Basic structure:

```
package <package_name>

import <another_class_1>;
…
import <another_class_N>;

public class <class_name>
{
        <class_body>
}
```

Classes

- Example:

```
package com.martin-toshev.examples

import com.martin-toshev.entities.Manufacturer;

public class Table
{
	private Manufacturer manufacturer;

	public Manufacturer getManufacturer() {
		return manufacturer;
	}

}
```

Packages

- Packages provide a mechanism to group classes logically
- Packages should follow the reverse naming convention in Java
- The fully qualified class name (FQCN) of a class is formed as <package_name>.<class_name>

```
com.martin-
toshev.examples.Table
```

Imports

- Imports provide a mechanism to import classes from other packages

- Either a single class or all classes from a package might be imported

```
import com.martin-
toshev.examples.Table;
import com.martin-toshev.examples.*;
```

Classes from the same and default packages do not need to be imported

The **java.lang** package is imported by default

Imports

- When a class is imported it can be used with its simple name

```
import com.martin-
toshev.examples.Table;
Table table = new Table();
```

- However a class might be used without being imported using its FQCN

```
com.martin-toshev.examples.Table table =
       new com.martin-toshev.examples.Table();
```

Imports

- Using FQCN instead of imports is useful in cases when there is a class with the same name in different packages

```
com.martin-toshev.examples.Table;
```

```
com.library.Table
```

- In that case one of the classes can be imported and the others referenced by FQCN

Multiple classes with the same FQCN coming from different libraries on the CLASSPATH of a Java application is problematic ! The JVM picks and uses just one of the classes in that case

There is a more general term coined in case of incompatible libraries on the CLASSPATH called **JAR hell**

Static Imports

- Since Java SE 1.5 the possibility to import static methods and constants has been provided

```
import static java.lang.Math.PI;
import static java.util.Collections.sort;
```

- All the static members of a class can be imported at once

```
import static java.util.Collections.*;
```

While static imports reduce amount of code written they can also provide ambiguity if used excessively (i.e. introduce confussion on whether the static method or constant is part of the class or not)

The CLASSPATH

- The CLASSPATH is simple a list of all the classes, directories and JAR files with classes that form the list of dependencies of the Java application

- The classpath is determined:
  - defaults to the current working directory of the application
  - entries in the CLASSPATH variable that override the previous
  - entries specified in the –cp (-classpath) command-line option that override the CLASSPATH variable
  - The runtime –jar option, which overrides the previous

Class body

- A class typically contains instance fields, instance methods and constructors
- However a class can also contain:
  - An init block
  - A static init block
  - Inner classes
- Let’s review each of them in details …

Instance fields

- Instance fields are typically the first thing to define in the class body

- To be more precise static fields (also called constants) are typically defined before instance fields

```
public class Table
{
        public static int SERIAL_NUMBER=123;

        private Manufacturer manufacturer;
        …
    }
```

Instance methods

- Instance methods follow typically after constants and instance fields in the class body

- The parts of an instance method is also called the **method signature** and constitutes of:
  - access modifier
  - return type
  - method name
  - parameter list
  - exception list

```
public boolean setTableColor(Color x)
    throws InvalidColorException,
        AnotherException
    {
    …
    }
```

Static fields

- Static fields (also called constants) are defined as instance fields but including the **static** keyword

- They are also called class variables as they are global to the class (i.e. shared among all objects of the class)

- Static fields might also be defined in interfaces

  The **static** keyword might be omitted when defining static fields in an interface

  It is a practice to use interfaces to define and hold a number of constants however it is preferable to use **enums** where applicable

Static fields

- Static fields can be referenced from the class directly

```
int serialNum = Table.SERIAL_NUMBER;
```

- They can also be accessed through instances of the class but that is discouraged (IDEs generate a warning)

```
Table table = new Table();
int serialNum =
     table.SERIAL_NUMBER; // generates a warning
```

Static fields cannot access instance fields and methods of a class

Static methods

- Static methods are defined as instance methods but using the **static** word in addition

- Static methods are global to the class (i.e. shared among the objects of the class)

```
public class Table
{
        public static int SERIAL_NUMBER=123;

        public static int getSerialNumber() {
                return SERIAL_NUMBER;
        }
        …
    }
```

Method overloading

- We already defined method overloading as defining multiple methods with the same name but different parameters in the same class

- There are certain rules involved in method overloading:

  - Overloaded methods must change the argument list
  - Overloaded methods can have a different return type
  - Overloaded methods can declare different exceptions being thrown
  - Overloaded methods can be defined in child classes

Method overloading

- Which one is called ?

```
public void sort(Collection collection) {
        …
}

public void sort(List list) {
        …
}
```

```
ArrayList arrayList = new ArrayList();
sort(arrayList);
```

Method overloading

- Which one is called ?

```
public void sort(Collection collection) {
        …
}

public void sort(List list) {
        …
}
```

```
ArrayList arrayList = new ArrayList();
sort(arrayList); // calls sort(List)
```

Method overloading

- What about these ?

```
public void sort(Cloneable list) {
        …
}

public void sort(List list) {
        …
}
```

```
ArrayList arrayList = new ArrayList();
sort(arrayList);
```

Method overloading

- What about these ?

```
public void sort(Cloneable list) {
        …
}

public void sort(List list) {
        …
}
```

```
ArrayList arrayList = new ArrayList();
sort(arrayList); // compile time error !
        // ArrayList implements both List and Cloneable
```

Method overloading

- What about these ?

```
public void sort(Cloneable list) {
        …
}

public void sort(List list) {
        …
}
```

```
ArrayList arrayList = new ArrayList();
sort((List) arrayList); // works fine
```

Constructors

- As we discussed constructors are special methods that do not return result and have the same class name
- They are used during object instantiation to provide initialization logic
- There are certain rules associated with constructors …

Constructors

- Constructors can use any access modifier
- A default constructor with no parameters is automatically generated by the compiler if none is provided
- In regard to the previous if you already have defined a constructor with parameters and want no-arg one you need to define it explicitly
- Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the superclass constructor (super())

Constructors

- If a constructor is defined and no call to super() or this() is made , the compiler will insert a no-arg call to super() as the very first statement in the constructor
- A call to an instance method or field cannot be made before the super constructor runs.
- Interfaces do not have constructors and they are not part of an object's inheritance tree
- The only way a constructor can be invoked is from within another constructor.

Interfaces

- Interfaces can define constants (static fields) and method signatures
- Interfaces do not participate in the object hierarchy but provide a way to reference objects with a restricted set of operations
- All interface methods are implicitly public

Interfaces

- As of JDK 8 interfaces can also have default and static methods !

```
public interface List {

        default void sort() {
                …
        }

        static void listTypes() {
                …
        }
}
```

Inner classes

- Classes can also contain other classes (so called **nested** or **inner** classes)

- A typical use of inner classes is to define helper types that are used only within the scope of the parent class

```
public class Table {

        class TableItem {
                …
        }
}
```

Inner classes

- Inner classes are instantiated by referencing them from an instance of the parent class

```
Table table = new Table();
Table.TableItem item = table.new TableItem();
```

While a parent class can have only a **public** or default access inner class can have any access modifier

There can be only one **public** top level Java class per Java source file but multiple top level classes with default access

Static inner classes

- Inner classes can also be static

- They are instantiated by referencing them from the parent class

```
public class Table {

        static class TableItem {
                …
        }
}
```

```
Table.TableItem item = new Table.TableItem();
```

Local inner classes

- Local classes can be define in the body of a method

```
public void move(Location location) {
        class LocationValidation {
                …
        }
}
```

Local classes are rarely used in practice and their use is discouraged !

Anonymous classes

- Anonymous classes provide a way to create and instantiate a class at the same time
- No name is specified for an anonymous class
- They make the code more compact

Anonymous classes

```
public void move(Location location) {
        new Thread(new Runnable() {

                @Override
                public void run() {
                    …
                }
        }).start();
}
```

As of Java 8 lambdas are the more preferable way to supply logic for an interface/abstract class with a single method rather than using an anonymous class

Inheritance

Why inheritance ?

- Inheritance is a fundamental concept of object-oriented programming

- Allows one class to extend the functionality of another class

- In Java one class can extend just one other class (but can implement multiple interfaces !)

Another form of inheritance is prototype-based as in Javascript where an object inherits functionality from another object (the **prototype**)

Extending classes

- A class extends another class using the **extends** keyword

```
public class RoundTable extends Table {

	public void move(Location location) {
		super.move(location);
		// logic here
	};
}
```

- If a class does not extend another class it implicitly extends from **java.lang.Object**

java.lang.Object

- It is generally discouraging to use java.lang.Object in places where you can use a more concrete type

```
// prefer using Table as a reference instead of Object !
Object table = new Table();
```

java.lang.Object

- It is generally discouraging to use java.lang.Object in places where you can use a more concrete type

```
// prefer using Table as a reference instead of Object !
Object table = new Table();
```

- java.lang.Object defines important methods that can be used on objects in Java

- Most of them are **native**: C++ implementation is provided by the JVM for the particular OS/CPU architecture

java.lang.Object

- java.lang.Object methods:
  - getClass(): retrieves the class of an object
  - hashCode()/equals(): very very important for HashMap !
  - toString(): returns the String representation of an object
  - clone(): used by classes implementing the Cloneable marker interface to provide support for cloning of objects
  - wait/notify methods: provide mechanism for communication between threads
  - finalize(): you should not override it

equals

- Two object in Java can be compared using the **equals** method

```
RoundTable roundTable = new RoundTable();
Table table = roundTable;
Table anotherTable = new Table();
System.out.println(table.equals(anotherTable)); // false
System.out.println(table.equals(roundTable)); // true
```

equals

- Unless equals is overridden the references are compared with ==

- Equals typically compares the fields of the two objects

  Never use == for comparison of objects (unless there is a specific reason to compare references) !

  By default **hashCode()** method (related to equals for use by HashMaps) returns object identity code (returned with **System.identityHashCode(<object>))**

instanceof

- We can an object is an instance of a particular class using **instanceof** operator

```
Table table = new Table();
System.out.println(table instanceof Table); // true
System.out.println(table instanceof RoundTable); // false
```

Avoid using **instanceof** as much as possible as it defies the principles of OOP (specifically polymorphism) !

Method overriding

- A recap: a child class can **override** (redefine) a method of the parent class which is called method overriding

- There are certain rules related to method overriding:
  - The argument list must exactly match that of the overridden method (otherwise we get an overloaded method)
  - the return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass
  - the access level can't be more restrictive than the overridden method’s
  - the access level can be less restrictive than that of the overridden method

Method overriding

- There are certain rules related to method overriding (continued):
  - A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final
  - A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass)
  - The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception
  - The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method

Method overriding

- There are certain rules related to method overriding (continued):
  - The overriding method can throw narrower or fewer exceptions
  - A method marked final cannot be overridden
  - Static methods cannot be overridden

Invalid overrides

- Examples:

```
public class Table {

        public void move() {…};
}
```

```
private void move() {…}
// invalid: more restrictive
```

```
private void move() throws Exception {…}
// invalid: throws undeclared in parent class
// checked exception !
```

```
private int move() {…}
// invalid: incompatible return type !
```

Superclass method

- A method in a child class can reference a method or a field in a parent class using **super**

```
public class RoundTable extends Table {

        public void move(Location location) {
                super.move(location);
                int color = super.color;
                …
        };
}
```

Constructors

- A constructor in the child class can call a particular constructor of the parent using **super(…)**

```
public class RoundTable extends Table {

	public RoundTable(Location location) {
		super(location);
	}
}
```

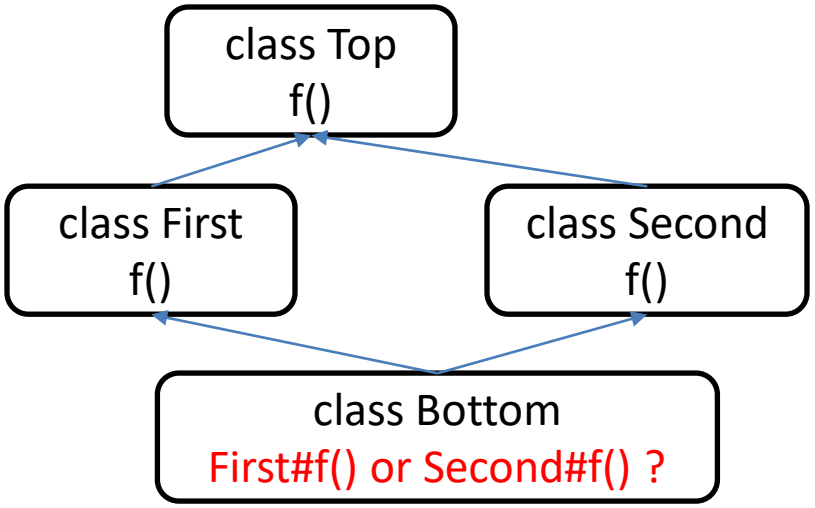
- If no call to **super(…)** is made then a call to **super()** is made implicitly

- A call to another constructor can be made with **this(…)**

Multiple inheritance

- Since multiple inheritance introduces complexity and in some cases ambiguity Java does not support it by design

- One of the most notorious problems related to multiple inheritance is the **diamond problem**



Implementing interfaces

- Interfaces are implemented using the **implements** keyword

- Multiple interfaces can be implemented (which is fine since they don't participate in the object hierarchy)

```
public class Table implements MovableTable, Cloneable {
        public void move(Location location) {
        // logic here
        }
}
```

Since JDK 8 interfaces have **default** methods which bring back the diamond problem, however the compiler makes additional checks to avoid it !

Instance members visibility

- A child class has access to the public and protected members of the parent class
- If the parent class is in the same Java package the Java class has also access to the fields with default access
- The child does not have access to the private members of the parent class

Be careful to avoid creating fields in the child class with the same name as those defined in a parent class as they shadow them !

Polymorhism

- Polymorphism refers to the concept of accessing different instances of child classes through a reference to a parent class

- The concept of **virtual methods** is also used in the context of polymorphism

- All instance methods of a class in Java that can be overridden (i.e. are not **private** or **final**) are virtual by design

Polymorhism

- A **virtual method** refers to the method of the particular class represented by the object that is called at runtime

```
Table table = new RoundTable();
table.move(); // calls RoundTable#move() regardless that
              // the reference is from the Table class !
```

- Polymorphic objects have the so called IS-A relationship (RoundTable IS-A Table)

- A different way to relate class not using inheritance is with instance fields or also called HAS-A relationship (Table HAS-A Location)

Type casting

- We already introduced type casting with primitive types
- Type casting can be also applied with classes:
  - **downcasting**: when casting from a parent to a child class (unsafe and needs a type check in some cases)
  - **upcasting**: when casting from a child to a parent class (safe and does not need type checking)

```
Table table = new RoundTable();
Object object = table; // upcasting
RoundTable roundTable = (RoundTable) table; // downcasting
```

Object oriented principles

The 4 pillars of OOP

- The four major principles of object oriented programming are:
  - abstraction: prefer abstractions rather than concrete classes
  - encapsulation: hide data in a class using accessor methods
  - Inheritance: classes can extend from other classes
  - polymorphism: classes can be varied using a parent reference

SOLID principles

- Fundamentals principles of OOP are further expanded by well-know patterns in the design of software systems

- The most notorious of these set of patterns are the SOLID principles:

  - **S**ingle responsibility principle
  - **O**pen-closed principle
  - **L**iskov substitution principle
  - **I**nterface segregation principle
  - **D**ependency inversion principle

Single responsibility principle

- **A class should have a single responsibility**

```
public class TableAndChair { // pretty bad !
    …
}
```

- Related to the principle of cohesion: the higher the cohesion of a class the more it implements a single responsibility

- A metric called lack of cohesion exists to try to determine statically if a class lacks cohesion or not

Open-closed principle

- **A class should be open for extension but closed for modification**

- Relates to encapsulation in terms of the fact that class data needs to be hidden

- Relates to inheritance and polymorphism in terms of the fact classes can extend the functionality of other classes

Liskov substitution principle

- **Objects can be replaced with instances of their child classes without affecting program correctness**

- In Java these principles are incorporated by the rules for method overriding we covered (which are enforced by the Java compiler)

Interface segregation principle

- **Many client-specific interfaces are better than one general-purpose interface**

- The principle guards the use of excessively large interfaces and abstract classes that force clients to implement a lot of methods (in many cases unnecessarily)

Dependency inversion principle

- **A class should depend on abstractions rather than concrete implementations**

- The principles is related to the coupling of a class: the less coupled a class is to other particular classes the better

- Since the principles enforces application to use a mechanism to **inject dependencies** in a class there are a number of dependency injection frameworks heavily used in practice

Questions ?