Best Practice in Developing Java Applications

Application Structuring with Maven. API & SPI

Dott. Marco Solinas, PhD

10 Nov 2021

Agenda

Maven

- What is Maven
- How Maven Works
- Parent POM
- Multi-module Maven Applications

API & SPI

- What is API?
- What is SPI?
- Event Listener and Asynchronous return
- API and SPI: always distinct concepts? The JDBC case
- The problem of code obfuscation
- Documenting public API with Javadoc

Application Structuring with Maven

Application Structuring with Maven

What is Maven

Maven is a tool for building and managing Java-based applications.
It is designed with the purpose of:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Encouraging better development practices

Source: http://maven.apache.org/what-is-maven.html

Application Structuring with Maven

How Maven works

Maven is based on the concept of Project Object Model (POM):

- There is a *pom.xml* file containing information about the project and configuration details used by Maven to build the project:
  - modelVersion: the version of the maven model
  - groupId: unique Id of the organization/group that created the project
  - artifactId: unique base name of the artifact generated by the project
  - version: version of the generated artifact
  - name: display name of this project
  - url: where this project can be found (useful for documentation)
  - properties: values whose scope is the current pom
  - dependencies: external libraries to be used in the project
  - build: contains build technical stuff, including the managing of plugins

Source: http://maven.apache.org/what-is-maven.html

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.mycompany.grp</groupId>
   <artifactId>my-app</artifactId>
   <version>1.0-SNAPSHOT</version>
   <name>My Application</name>
   <url>http://www.myappexample.com</url>

   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.junit.jupiter</groupId>
         <artifactId>junit-jupiter-api</artifactId>
         <version>5.6.2</version>
         <scope>test</scope>
      </dependency>
   </dependencies>

   <build>
      <pluginManagement>
          ... lots of helpful plugins
      </pluginManagement>
   </build>
</project>
```

Application Structuring with Maven
Parent POM

A pom file can inherit from a ‘super-pom’, a.k.a. parent pom

- Project **Inheritance**
  - The parent pom can be the pom of another project or even a pom defined at organization level
  - In child pom, **groupId** and **version** override the values that otherwise would be inherited from the parent

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
   <modelVersion>4.0.0</modelVersion>

   <groupId>it.unipi.dii</groupId>
   <artifactId>dii-parent</artifactId>
   <version>1</version>
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
   <modelVersion>4.0.0</modelVersion>

   <parent>
      <groupId>it.unipi.dii</groupId>
      <artifactId>dii-parent</artifactId>
      <version>1</version>
   </parent>

   <groupId>it.unipi.dii</groupId>
   <artifactId>library-main</artifactId>
   <version>1</version>
…
</project>
```

- Project **Aggregation**
  - The parent pom knows its modules:
    - **packaging**: pom
    - **modules** section with sub-modules artifactIds
  - Any mvn command ran against the parent is also ran against all his modules

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
   <modelVersion>4.0.0</modelVersion>

   <groupId>it.unipi.dii</groupId>
   <artifactId>library</artifactId>
   <version>1</version>
   <packaging>pom</packaging>

   <modules>
      <module>library-main</module>
   </modules>
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
   <modelVersion>4.0.0</modelVersion>

   <parent>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library</artifactId>
      <version>1</version>
   </parent>

   <groupId>it.unipi.dii</groupId>
   <artifactId>library-main</artifactId>
   <version>1</version>
…
</project>
```

Source: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Application Structuring with Maven

Multi-module Maven Applications (1/2)

Project Aggregation can be exploited to build multi-module applications:

- Code base of an application can significantly grow
  - The longer the application lasts, the huger the amount of code
- Split the codebase into many modules, each representing a specific concern of your application domain
  - Modules can refer to each other in their poms' **dependencies**
  - Mind the circular dependencies!
- Parent pom should contain
  - Common set of third-party dependencies all the plugins use
    - The **dependencies** section must go into the **dependencyManagement** section
  - Common Maven properties
  - Common Maven plugins definitions used for building the Maven modules
    - Note the SNAPSHOT version (maven release plugin)

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>library-main</module>
    <module>library-common</module>
    <module>library-books</module>
    <module>library-music</module>
  </modules>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>it.unipi.dii</groupId>
    <artifactId>library</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library-main</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library-common</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library-music</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit-jupiter-api</artifactId>
    </dependency>
  </dependencies>
…
</project>
```

Application Structuring with Maven

Multi-module Maven Applications (2/2)

Project Aggregation can be exploited to build a common and coherent set of shared libraries:

- Why common? Why coherent?
- Each module represents a library
  - Modules can still refer to each other
- Parent pom can be used as parent for any application
  - Project inheritance!

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <groupId>it.unipi.dii</groupId>
  <artifactId>dii-parent</artifactId>
  <version>12.7.34</version>
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>it.unipi.dii</groupId>
    <artifactId>dii-parent</artifactId>
    <version>12.7.34</version>
  </parent>

  <groupId>it.unipi.dii.inginf</groupId>
  <artifactId>ing-inf-parent</artifactId>
  <version>2.0.4</version>
  <packaging>pom</packaging>

  <modules>
    <module>cache</module>
    <module>login-utils</module>
    <module>rmi-pisa</module>
    <module>pub-sub-pisa</module>
  </modules>
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>it.unipi.dii.inginf</groupId>
    <artifactId>ing-inf-parent</artifactId>
    <version>2.0.4</version>
  </parent>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packag

  <modules>
    <module>library-main
    <module>library-comm
    <module>library-book
    <module>library-musi
  </modules>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jun
        <artifactId>juni
        <version>5.6.2</
        <scope>test</sco
      </dependency>
      …
    </dependencies>
  </dependencyManagement
…
</project>
```

```
<project xmlns="…" xmlns:xsi="…"
         xsi:schemaLocation="…">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>it.unipi.dii</groupId>
    <artifactId>library</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library-main</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library-common</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>it.unipi.dii.inginf</groupId>
      <artifactId>cache</artifactId>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
    </dependency>
  </dependencies>
…
</project>
```

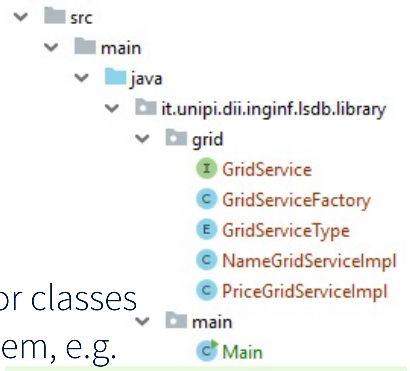
API & SPI

API & SPI

What is API?

Application Programming Interface:

- Publicly accessible set of items (interfaces and/or classes and/or methods…) provided by an external system, e.g.
  - In a library we imported in our application
  - Of a remote service we must invoke through the network
- Call and use what is available in an API to have access to the functionalities required in an application
- Isolate public items (e.g. interface, shared beans…) from implementation details
  - Exploit package visibility for objects that are NOT public
  - Good practice: put implementation classes in a sub-package
- Addition is not a problem, but Removal is
  - @Deprecated



```
src
  main
    java
      it.unipi.dii.inginf.lsdb.library
        grid
          GridService
          GridServiceFactory
          GridServiceType
          NameGridServiceImpl
          PriceGridServiceImpl
        main
          Main
```

API

```java
package it.unipi.dii.inginf.lsdb.library.grid;
import java.util.List;
public interface GridService {
    List<String> getColumnNames();
}
```

API

```java
package it.unipi.dii.inginf.lsdb.library.grid;
public enum GridServiceType {
    NAME, PRICE;
}
```

impl

package scope!

```java
package it.unipi.dii.inginf.lsdb.library.grid;
import com.google.common.collect.Lists;
import java.util.List;
class PriceGridServiceImpl implements GridService {
    PriceGridServiceImpl() {}
    public List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

API

private scope!

```java
package it.unipi.dii.inginf.lsdb.library.grid;
public class GridServiceFactory {
    private GridServiceFactory() {}
    public static GridServiceFactory create() {
        return new GridServiceFactory();
    }
    public GridService getService(GridServiceType type) {
        switch (type) {
            case NAME:
                return new NameGridServiceImpl();
            case PRICE:
                return new PriceGridServiceImpl();
        }
        return null;
    }
}
```

API & SPI

What is SPI? (1/2)

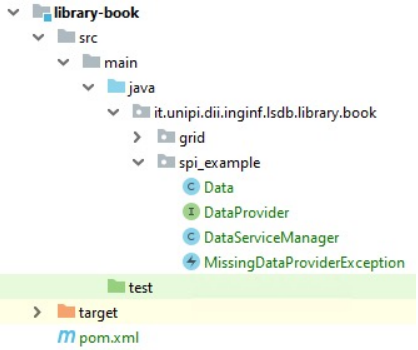
Service Provider Interface:

- Publicly accessible set of items that can be implemented or extended to achieve a goal, e.g.
  - In a library we imported in our application

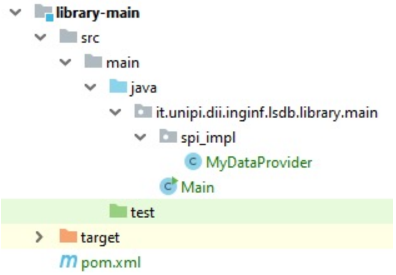
- Application is responsible for the implementation of a specific interface invoked by the library

- SPI Interfaces can be invoked to
  - Allow the computation flow of the library to proceed, e.g.: a data provider
  - Modify the behavior of an imported library to meet application requirements, e.g.: price formatter
  - Notify the application of a particular condition/event, e.g.: a data availability listener

- Removal is not a problem, Addition is



```
library-book
  src
    main
      java
        it.unipi.dii.inginf.lsdb.library.book
          grid
          spi_example
            Data
            DataProvider
            DataServiceManager
            MissingDataProviderException
    test
  target
  pom.xml
```



```
library-main
  src
    main
      java
        it.unipi.dii.inginf.lsdb.library.main
          spi_impl
            MyDataProvider
          Main
    test
  target
  pom.xml
```

API & SPI

What is SPI? (2/2)

API

```java
package it.unipi.dii.inginf.lsdb.library.book.spi_example;
public class DataServiceManager {

    private static DataServiceManager instance;
    private DataProvider dataProvider;

    private DataServiceManager(){}

    public static DataServiceManager getInstance() {  // it's a singleton instance
        if (instance == null) {
            instance = new DataServiceManager();
        }
        return instance;
    }

    public void registerDataProvider(DataProvider dataProvider) {
        this.dataProvider = dataProvider;
    }

    public Data catenate(String dataId1, String dataId2) throws MissingDataProviderException {
        if (dataProvider != null) {
            Data data1 = dataProvider.getDataById(dataId1);
            Data data2 = dataProvider.getDataById(dataId2);
            Data resultData = null;
            if (data1 != null && data2 != null) {
                resultData = doCatenate(data1,data2);
            }
            return resultData;
        }
        throw new MissingDataProviderException();
    }
}
```

implementation is delegated to the application

very important

SPI

```java
package it.unipi.dii.inginf.lsdb.library.book.spi_example;
public interface DataProvider {
    Data getDataById(String dataId);
}
```

impl

```java
package it.unipi.dii.inginf.lsdb.library.main.spi_impl;

import com.google.common.collect.Maps;
import it.unipi.dii.inginf.lsdb.library.spi_example.Data;
import it.unipi.dii.inginf.lsdb.library.spi_example.DataProvider;

import java.util.Collection;
import java.util.Map;

public class MyDataProvider implements DataProvider {

    private final Map<String, Data> dataCache = Maps.newHashMap();

    public MyDataProvider() { }

    @Override
    public Data getDataById(String dataId) {
        return dataCache.get(dataId);
    }

    public void addNewData(Data data) {
        dataCache.put(data.getId(), data);
    }

    public void addNewData(String value) {
        addNewData(new Data(value));
    }

    public Collection<Data> getAll() {
        return dataCache.values();
    }

}
```

API & SPI

Event Listener and Asynchronous return (1/2)

API

```
package it.unipi.dii.inginf.lsdb.library.book.spi_example;
import com.google.common.collect.Lists;
import java.util.Collection;

public class DataServiceManager {

    private static DataServiceManager instance;
    private DataProvider dataProvider;
    private Collection<DataAvailabilityListener> listeners = Lists.newArrayList();

    private DataServiceManager(){}

    public static DataServiceManager getInstance() { … }

    public void registerDataProvider(DataProvider dataProvider) { … }

    public void registerDataAvailabilityListener(DataAvailabilityListener listener) {
        if (listener != null) listeners.add(listener);
    }

    public CompletableFuture<Data> catenateAsyncronously(String dataId1, String dataId2) {
        CompletableFuture<Data> retval = new CompletableFuture<>();

        if (dataProvider != null) {
            Data data1 = dataProvider.getDataById(dataId1);
            Data data2 = dataProvider.getDataById(dataId2);

            if (data1 != null && data2 != null) {
                scheduleNewThreadOnProperExecutor(() -> {
                    Data resultData = doCatenate(data1, data2);
                    listeners.forEach(l -> {
                        l.onDataAdd(resultData);
                    });
                    retval.complete(resultData);
                });
            }
        } else {
            retval.completeExceptionally(new MissingDataProviderException());
        }
        return retval;
    }
}
```

doCatenate might be a heavy operation → execute it on a separate thread

SPI

```
package it.unipi.dii.inginf.lsdb.library.book.spi_example;
public interface DataAvailabilityListener {
    void onDataAdd(Data d);
    void onDataRemove(Data d);
}
```

impl

```
package it.unipi.dii.inginf.lsdb.library.book.main.spi_impl;

import it.unipi.dii.inginf.lsdb.library.book.spi_example.Data;
import it.unipi.dii.inginf.lsdb.library.book.spi_example.DataAvailabilityListener;

public class MyDataListener implements DataAvailabilityListener {
    private final MyDataProvider provider;
    public MyDataListener(MyDataProvider provider) {
        this.provider = provider;
    }
    @Override
    public void onDataAdd(Data d) {
        provider.addNewData(d);
    }
    @Override
    public void onDataRemove(Data d) {
        // do nothing
    }
}
```

Event Listener and Asynchronous return (2/2)

```
public class Main {

    private static Scanner reader = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        DataServiceManager manager = DataServiceManager.getInstance();

        // we must register a data provider and a data listener
        MyDataProvider provider = new MyDataProvider();
        manager.registerDataProvider(provider);
        manager.registerDataAvailabilityListener(new MyDataListener(provider));

        char c = '0';
        do {
            displayMenu();
            c = reader.nextLine().charAt(0);
            switch (c) {
                case '1':
                    doDisplayAll(provider);
                    break;
                case '2':
                    doAdd(provider);
                    break;
                case '3':
                    doCatenate(manager, provider);
                    break;
                default:
                    if (c != '0') {
                        System.out.println("Invalid option\n\n");
                    }
            }
        } while (c != '0');
    }

    private static void displayMenu() { … }

    private static void doDisplayAll(MyDataProvider provider) { … }

    private static doAdd(MyDataProvider provider) { … }
```

```
    private static void doCatenate(DataServiceManager manager,
                                   MyDataProvider provider) {
        System.out.print("Give me the first Id: ");
        String id1 = reader.nextLine();
        System.out.print("Give me the second Id: ");
        String id2 = reader.nextLine();

        try {
            Data dataRes = manager.catenate(id1, id2);
            System.out.println("Concatenated Data: " + dataRes.getValue());
            provider.addNewData(catenated);
        } catch (MissingDataProviderException e) {
            e.printStackTrace();
        }
    }

    private static void doCatenateAsync(DataServiceManager manager,
                                        MyDataProvider provider) {
        System.out.print("Give me the first Id: ");
        String id1 = reader.nextLine();
        System.out.print("Give me the second Id: ");
        String id2 = reader.nextLine();

        CompletableFuture<Data> future = manager.catenateAsyncronously(id1, id2);
        future.whenComplete((dataRes, throwable) -> {
            if (throwable != null) {
                System.out.println("Concatenated Data: " + dataRes.getValue());
            } else {
                throwable.printStackTrace();
            }
        });
    }

} // END OF CLASS MAIN
```

API and SPI: always distinct concepts? The JDBC case

JDBC is a set of specifics, not an implementation nor a library.

- Many vendors implement the JDBC specifics (they must be JDBC-compliant).
- Publicly accessible items can be part of the API and/or the SPI
- The **Driver** class is an example of pure SPI item
  - You don’t need to use it directly in an application, but vendors must implement it
- The **Connection** interface is an item that is both API and SPI
  - It is invoked in the application and must be implemented by the vendors

API & SPI

The problem of code obfuscation (1/2)

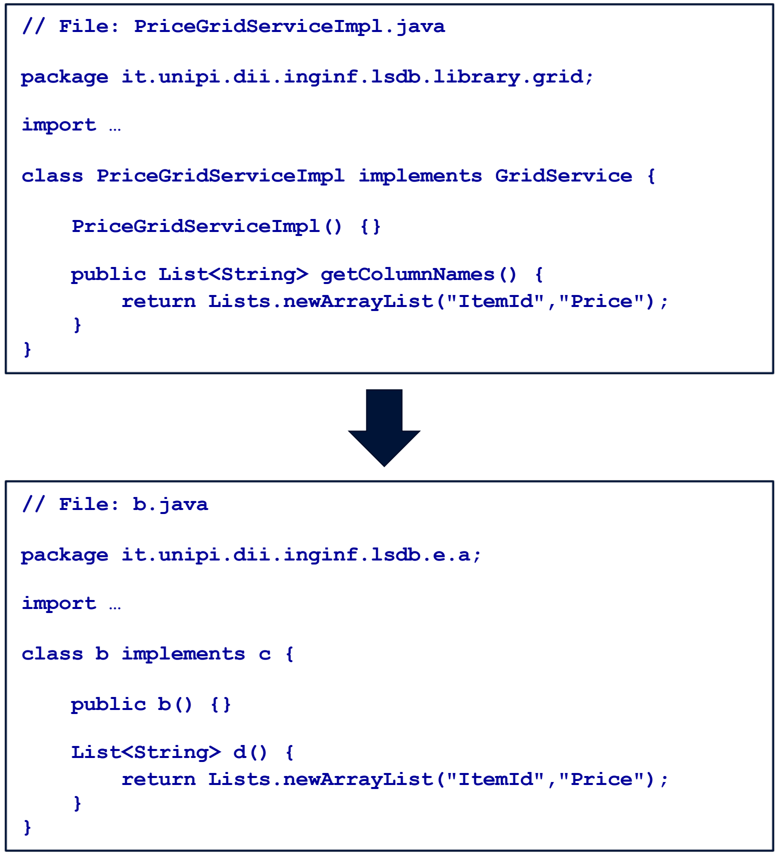
It is always possible to decompile the java bytecode and get the source code.
To avoid this, companies often adopt java code obfuscation

- Typically, a preliminary parsing of the files is done
  - files/classes/methods/fields are renamed with random identifiers
  - All comments are (typically) removed
  - E.g., ProGuard: https://www.guardsquare.com/en/products/proguard

- If a malicious user tries to decompile the distributed .class files, he will get a set of java files and classes with unmeaningful names

- Within an application this is not a problem, the obfuscator will rename all the references in a coherent manner

- Problem: libraries and multi-module maven application



```
// File: PriceGridServiceImpl.java

package it.unipi.dii.inginf.lsdb.library.grid;

import …

class PriceGridServiceImpl implements GridService {

    PriceGridServiceImpl() {}

    public List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

```
// File: b.java

package it.unipi.dii.inginf.lsdb.e.a;

import …

class b implements c {

    public b() {}

    List<String> d() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

API & SPI

The problem of code obfuscation (2/2)

It is always possible to decompile the java bytecode and get the source code.
To avoid this, companies often adopt java code obfuscation

- When the items we want to refer are not in the same module of our code, they cannot be obfuscated
- Decoupling the public API/SPI from the implementation helps
  - Publicly accessible classes/interfaces are kept
  - Underlying implementation can be obfuscated
- Common obfuscation tools provide methods to tell the parser to keep specific items clear
  - E.g., ProGuard has a set of *@Keep** annotations

```
package it.unipi.dii.inginf.lsdb.e.a;
import …;

@KeepName
@KeepPublicClassMemberNames
public interface GridService {
    List<String> getColumnNames();
}
```

```
// File: PriceGridServiceImpl.java

package it.unipi.dii.inginf.lsdb.library.grid;

import …

class PriceGridServiceImpl implements GridService {

    PriceGridServiceImpl() {}

    public List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

```
// File: b.java

package it.unipi.dii.inginf.lsdb.e.a;

import …

class b implements GridService {

    public b() {}

    List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

API & SPI

Documenting the public API with Javadoc

It is a good practice to document the public API using Javadoc.

- Should be readable as source code
  - Both for your teammates and for third-part library

- Public and protected methods should be fully documented
  - Indeed, private and package methods can still benefit
  - Overridden methods: only if the redefinition has a different behavior

- Use plain HTML tags, e.g. <p> <br/> <ul> and <li>

- A few interesting Javadoc tags commonly used
  - @param, @return and @throws
  - @link and @code
  - @since and @see

Source: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

```
/**
 * Manager for Multimedia objects.
 * @since  2.4.1
 * @see   {@link Multimedia}
 */
public class MultimediaManager {
   //… private part

   /**
    * Processes a new Multimedia and its price.
    * <p>
    * Depending on the price value:
    * <ul>
    * <li>p>0: we do this and not that</li>
    * <li>p=0: we do that and not this</li>
    * <li>p<0: we do both this and that</li>
    * </ul>

    * @param m  the Multimedia to be processed, not null
    * @param p  the Multimedia price
    * @throws {@link MultimediaException} if m is null or invalid
    */
   public void process(Multimedia m, double p) throws MultimediaException {
      //…
   }

   /**
    * Reads a Multimedia provided its unique identifier

    * @param id  the unique identifier of the Multimedia
    * @return the Multimedia corresponding to the id, null if not found
    */
   public Multimedia getMultimedia(String id) {
      //…
   }
}
```