Advanced Java Programming

Kitlei Róbert

Department of Programming Languages and Compilers
ELTE Faculty of Informatics

Java compilation

- .java files are compiled to .class by the compiler
  - ◇ What if there are ***too many*** of them?
  - ◇ We need to use external ***libraries***
  - ◇ We need to run ***tests***
- Solution: ***build tools*** (with ***package managers***, also called ***dependency managers***)
  - ◇ Maven
  - ◇ Ant+Ivy
  - ◇ Gradle
  - ◇ jpm4j

Maven configuration

- ***POM***: Project Object Model (contained in `pom.xml`)
  - ◇ Describes the project configuration
  - ◇ ***GAV***: groupId:artifactId:version
  - ◇ Packaging: the result of the build (pom, jar, war, ear, ...)

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>hu.elte.inf.kitlei</groupId>
    <artifactId>mypackagename</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
</project>
```

- The project directory structure is fixed
  - ◇ `src/main`: contains `/java`, `/webapp` and `resources`
  - ◇ `src/test`: contains `/java` and `resources`
  - ◇ `target`: output directory

Maven POM

- POMs can control many projects
  - ◇ each will have a `pom.xml` in their directories

```
<project>
    ...
    <packaging>pom</packaging>
    <modules>
        <module>project1</module>
        <module>project2</module>
    </modules>
</project>
```

Maven Build Phases

- ***Build Lifecycle***: When the project is built, the following steps are run
  - ◇ `validate`, `initialize`
  - ◇ `generate-sources`, `generate-resources` (with `process-`)
  - ◇ `compile` (with `process-classes`)
  - ◇ `test` (with some more related steps)
  - ◇ `package` (with `prepare-`)
  - ◇ `integration-test` (with `pre-` and `post-`)
  - ◇ `verify`
  - ◇ `install`
  - ◇ `deploy`
- ***Goal***: Invoke these steps (and everything above it) by executing

  `mvn <phase name>`

  - ◇ There is also `mvn clean` (with `pre-clean`, `post-clean`)
  - ◇ Set more goals one after the other: `mvn clean compile`

Dependency management

- If your project needs a library, Maven finds it
  - ◇ Transitive: if your library needs a library (which needs a library etc.), Maven still finds them and downloads them
  - ◇ ... provided that they are in the ***Maven Central Repository***
  - ◇ Alternatively, you can setup a ***Proxy Repository***
- The required library's GAV has to be added to the POM

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>com.testsite</groupId>
            <artifactId>program-tester</artifactId>
            <version>1.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies></project>
```

- More: exclude, optional, manual clash resolution

Other similar tools

- Ant: build system, Ivy: dependency manager
  - ◇ More flexible, e.g. directory structures are not fixed
  - ◇ Compatible with Maven repositories
- Gradle
  - ◇ uses Groovy, a language similar to Java

Jar files

- Libraries in Java are usually `.jar` files
- They are zip files, usually created with the `jar` tool
- They contain:
  - `META-INF/MANIFEST.MF`: metadata
    - Automatically created if unspecified
  - `.class` files: bytecode for the JVM
    - They have to be placed in a directory hierarchy
    - The directory hierarchy has to respect the package hierarcy
    - E.g. `abc.def.hij.Xyz` has to go in `Xyz.class` in `abc/def/hij` inside the archive

The jar tool

- Similar to tar
  - ◇ ***c***reate, e***x***tract, ***u***pdate
  - ◇ ***v***erbose, from ***f***ile, include ***m***anifest

```
jar cvf  test.jar          hu/elte/inf/*.class
jar cvfm test.jar test.mf hu/elte/inf/*.class
```

- The manifest can contain things like:

```
Classpath: ./test.jar
Main-Class: Test
```

- Jar files can be added to the classpath manually

```
java -cp .:mylibrary.jar LibraryUser
```

Class loading

- Classes are usually stored locally
- ... but they can be dynamically loaded
  - ***Java Web Start***: start a Java program from a web page
  - ***Proxies*** can be downloaded from naming services
    - They contain code to access remote services

```
                    //  String      boolean
Class r = loadClass(className, resolveIt);
```

- `resolveIt`: Should the referenced classes be loaded as well?
- When are classes loaded?
  - When bytecode from it has to be executed (e.g. `new MyClass();`)
  - When bytecode statically refers to it (e.g. `System.out`)

Using class loaders

- The virtual machine uses the ***primordial*** class loader to start the program
  - ◇ It loads `java.lang.Object`
  - ◇ It knows some ***trusted classes***
  - ◇ It can be replaced by a custom one
    - ▸ Subclass of `java.lang.ClassLoader`, only method: `loadClass`
- Usual tasks of `loadClass`
  - ◇ Verify class name
  - ◇ Check to see if the class requested has already been loaded
  - ◇ Check to see if the class is a system class
  - ◇ Attempt to fetch the class from this class loader's repository
  - ◇ Define the class for the VM
  - ◇ Resolve the class
  - ◇ Return the class to the caller

Custom class loader

```
Map<String, Class> loadedClasses;
public synchronized Class loadClass(String className,
                                    boolean resolveIt)
        throws ClassNotFoundException {
    byte classData[];

    Class cached = loadedClasses.get(className);
    if (cached != null)    return cached;
    try {
        // try to get class from classpath
        return super.findSystemClass(className);
    } catch (ClassNotFoundException e) {
        System.out.println("Not a system class.");
    }
}
```

Issues

- Loaded classes can be a security hazard
  - ◇ A class in a sensitive package, e.g. `java.lang`, can access critical data
  - ◇ Careless handling of the `className` argument can lead to bad things
- Instances of the loaded class cannot be cast to their proper types
  - ◇ Only a cast to a trusted class/interface is allowed
  - ◇ So a base class or interface (loaded by the primordial class loader) has to be used

```
CustomClassLoader ccl = new CustomClassLoader();
Class  c = ccl.loadClass("MyCl.txt");
Object o = c.newInstance();
((MyCl)o).myMethod();      // invalid, only ccl knows of MyCl
IFace ifc = (IFace)o;      // common solution supposing that
                           // MyCl implements this interface
((IFace)o).myIFaceMethod();   // this is OK, too
```