CSE 1325 - Object-Oriented Programming

Organizing Projects

Alex Dillhoff

University of Texas at Arlington

Organizing Projects

Overview

1. Packages
2. Creating JAR Files
3. Running JAR Files

Organizing Projects

Java provides the ability to organize multiple related class files within a `package`.

We have already used packages with some of the Java API classes, such as `java.time.LocalDate`.

Organizing Projects

When creating a package, it is important to choose a unique name.

Textbooks and other resources frequently suggest a domain name scheme.

For example, the domain `uta.edu` would be reversed as a package name.

Organizing Projects

We may then choose to create a project named `CSE1325` and place all of our relevant class files into the package `edu.uta.CSE1325`.

A class that is part of a package may reference any other class within that package.

Organizing Projects

Classes may also be accessed specifically. For example:

```
java.time.LocalDate today = java.time.LocalDate.now();
```

However, this is very verbose and it is more convenient to import the packages instead.

Adding a Class to a Package

Adding a new class to an existing package is simple. For example, if we want to add a new class to the `edu.uta.CSE1325` package, we simply add the line

```
package edu.uta.CSE1325;
```

to the top of the new class file.

Adding a Class to a Package

Let's look at a more detailed example project using multiple classes.

Example: `ShoppingApp`

Shopping App

Our task is to create a simple shopping system in which a **customer** can view **items**, add them to their **cart**, and **order** them.

- **Which classes should we create?**
- **What actions should they have?**
- **How do we integrate them together?**

Organizing Classes

When using classes from a particular package, the directory path they are stored in must match the package name.

In the previous example, the new class in package `edu.uta.CSE1325` is stored in the path `./edu/uta/CSE1325`

Organizing Classes

When you compile a Java program, the compiler will look for classes in directories defined by the `classpath`.

By default, the `classpath` is set to the current directory (`'.'`).

Organizing Classes

If a class in your base directory imports the package `edu.uta.CSE1325`, it does not need to be explicitly added to the `classpath`.

The compiler will look for the matching path relative to the `classpath`, in this case: `./edu/uta/CSE1325`

The `classpath`

Additional paths can be added to the `classpath`. When you do this, the `classpath` will look something like this:

```
/home/student/projects:.:/home/student/archive
```

Each path is separated by `:`, and the compiler will look through all paths when compiling.

The `classpath`

If the `"."` is not present in the `classpath`, then the compiled class files that are compiled may not be included when running the program.

This results in programs that will compile but will not run.

The `classpath`

Setting the `classpath` can be done when compiling a Java program by using one of the following flags:

- `-classpath`
- `-cp`
- `--class-path`

The classpath

Example: Setting `classpath` on UNIX

```
java -classpath /home/student/projects:. Player
```

The classpath

Example: Setting `classpath` on Windows

```
java -classpath c:\Users\student\projects;. Player
```

Note that the paths on Windows are separated by `";"` instead of `":"`.

The classpath

Alternatively, the CLASSPATH environment variable can be set in either Windows or UNIX.

Java Archives (JAR)

You can imagine that a large project with many class files would be cumbersome to share with other users.

Fortunately, there is a solution: **Java Archives (JAR)**.

Java Archives (JAR)

JAR files use ZIP compression to store multiple class files along with other relevant assets (video, sound, etc.).

Any project can be packaged into a JAR file and any JAR file can be included into your own project by adding it to the `classpath`.

Java Archives (JAR)

The prototype for the `jar` command is as follows:

```
jar [OPTION ...] [[--release VERSION] [-C dir] files] ...
```

Java Archives (JAR)

A common usage of `jar` will look something like:

```
jar cvf MyProject.jar *.class
```

The compression options will be familiar for those who have use `tar` to create archives.

The JAR Manifest

A JAR file also contains a **manifest** which describes its contents.

A default manifest is added to every JAR file by default. Additional information can be added via a text file.

The JAR Manifest

As an example, let's say we want to add information about the `Player class`.

```
Manifest-Version: 1.0
Table Top RPG Game

Name: Player.class
Contains behaviors defining individual players.
```

The JAR Manifest

The manifest can then be included with any JAR file. For example,

```
jar cfm JarFile.jar ManifestFile.mf
```

The JAR Manifest

To update the manifest of an existing archive, use the `u` option instead of `c`.

```
jar ufm JarFile.jar ManifestFile.mf
```

The JAR Manifest

For more information, see the tutorial here:

`https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html`

Executable JAR Files

It is possible to create a single executable for your project in the form of a Java archive.

This is useful for packaging large projects that can be run on any system with the JRE installed.

Executable JAR Files

This is done easily by adding the e option when creating the JAR file.

```
jar cvfe JarFile.jar MainFile AdditionalFiles
```

Executable JAR Files

You can also specify the main class of the program as part of the manifest file.

```
Main-Class: edu.uta.Package.MainClass
```

Executable JAR Files

Example: JetUML