18

Final Submission and Essay

CERTIFICATION OBJECTIVE

- Preparing the Final Submission

CERTIFICATION OBJECTIVE

Preparing the Final Submission

You've built your project, and now the Big Day is finally here. Submission time. Your exam instructions include very specific details for submission, and you must follow them *exactly*. *Pay attention: any deviation from the submission instructions can mean automatic failure.* For example, if your instructions say that you must be able to run your application with a specific command,

```
java -jar runme.jar server
```

you had better have a JAR named `runme.jar`, and it better take a command-line argument "server", and it better include a manifest that specifies the class within `runme.jar` that holds the `main()` method.

In this short chapter we'll look at a typical submission requirement, and walk through how to build the final JAR along with a project checklist. Finally, we'll look at some examples of the kinds of essay questions you might see on your follow-up exam.

File Organization

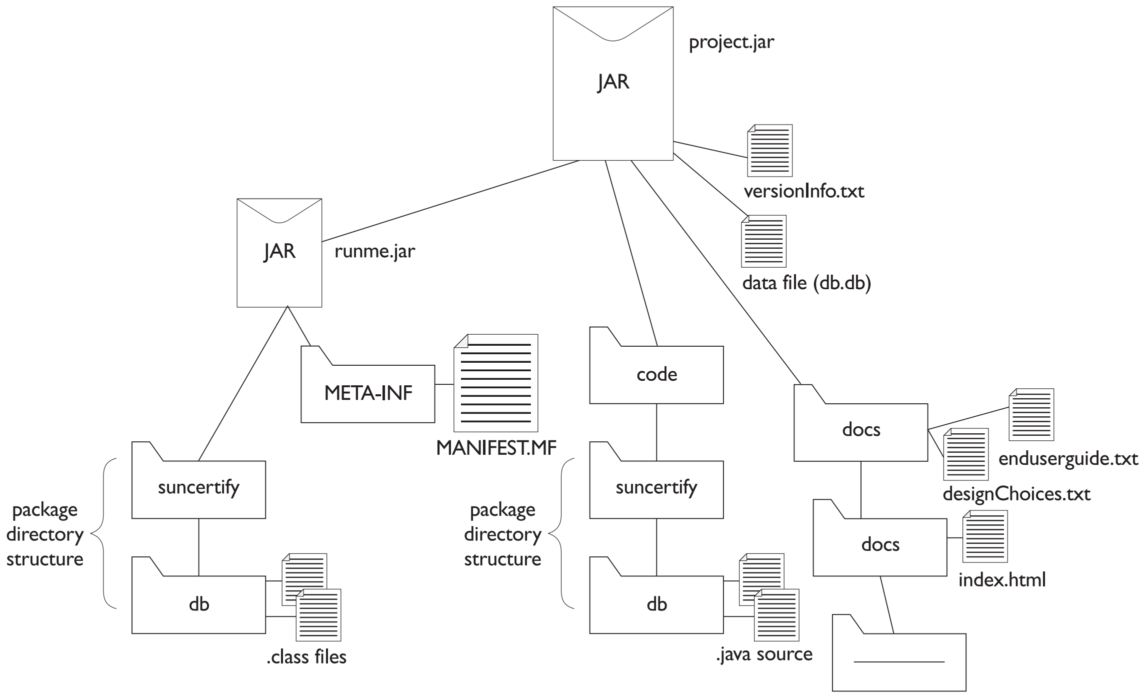
Imagine the following submission instructions; yours will probably be very similar:

- All project files must be delivered in one, top-level Java Archive (JAR) file.
- The top-level, project JAR must be named **project.jar**.
- The project JAR must contain the following files and directories:
  - An *executable* JAR named **runme.jar** that contains the complete set of classes.
  - The **code** directory, which must hold the source code for your project, with all source files organized within directories reflecting the package structure of the classes.
  - A version file named **versionInfo.txt**. This must be a plain ASCII text file describing the specific version of J2SDK that you used (example: java version "1.3.1").
  - A copy of the data file, exactly as specified in the schema instructions, named **db.db**.

- The **docs** directory, which must hold all project documentation including:
  - A design decision document named **designChoices.txt**, an ASCII text file documenting design decisions.
  - End-user documentation for the server and client, unless you have used an online help system within your application. The help documents may consist of multiple HTML files but must begin with an HTML file user guide.
  - *javadoc* HTML files for all classes and interfaces. All public classes, interfaces, and members must be documented.
  - Developer documentation, optional.

Figure 18-1 illustrates the directory structure that matches the sample instructions above.

FIGURE 18-1 Sample directory structure for project submission



Creating the Executable JAR

An executable JAR is a JAR file that contains at least two things:

- A class file with a `main()` method.
- A manifest file that specifies *which* class in the JAR has the `main()` method. (Remember, you might have dozens of classes in your application JAR file.)

Creating the Manifest You can let the *jar* tool create both a manifest file (`MANIFEST.MF`) and the manifest directory (`META-INF`), but you'll need to put your information into the manifest file. The *jar* tool has a command that lets you specify your own text file as a place to find text that will be merged into the *real* manifest file. In other words, you won't actually *create* the manifest file yourself but you'll build a text file that has the stuff you want to *add* to the "official" manifest file the *jar* tool will create. This is not the only way to make your manifest file, but it's usually the simplest, and the end-result is the same: the `MANIFEST.MF` file will declare that you have a class with a `main()` method.

Assume the following structure for the examples we're going to walk through:

- **Main class name**, `suncertify.db.ExamProject` You have a class named ExamProject, in the `suncertify.db` package, and it holds the `main()` method for your application.
- **Working directory**, `/project` This is the directory one level *above* your package structure.
- **Manifest file name**, `Manifest.MF` This is not the *real* manifest file but rather your text file that holds the text you want to merge into the *real* manifest file that the *jar* tool will create.
- **Manifest file contents** The manifest file you'll write (as an ASCII text file) contains just a single line:
  ```
  Main-Class: suncertify.db.ExamProject
  ```

exam watch

Be certain to insert a carriage return at the end of that single line! If there is not a *newline* ***below the main line, the manifest file will not work correctly. And be certain to include the*** *fully-qualified name* ***of the main class (***`suncertify.db.ExamProject` ***as opposed to just*** `ExamProject`***).***

Creating the JAR We've got our application classes in their appropriate directory structure (matching the package structure). We've got our manifest file, `Manifest.MF`. We're in a working directory that is one level *above* the package, so if you listed the contents of the current directory it would include a directory named `suncertify` (the first directory in the package hierarchy).

So let's run the *jar* tool:

```
jar -cvmf Manifest.MF runme.jar suncertify/db
```

This creates a JAR file, named `runme.jar`, in your current directory. It includes all files that are in the `suncertify/db` directory (the package) and it also includes the directories themselves (`suncertify` and `db`). Plus, it takes the manifest file (`Manifest.MF`), reads its contents, and puts them into the *real* manifest file named `MANIFEST.MF,` that it puts in a directory named META-INF. At the command-line, you'll see the following when you run the `jar` command as follows:

```
jar -cvmf Manifest.MF runme.jar suncertify/db
added manifest
adding: suncertify/db/(in = 0) (out= 0)(stored 0%)
adding: suncertify/db/ExamProject.class(in = 617) (out= 379)(deflated 38%)
```

Of course in your *real* project, you'll have more than one class. Using the preceding `jar` command, you'll see the manifest being added, the directory structure being added (in this case, `suncertify/db)`, and all files in the `db` directory (in this case, just the `ExamProject.class`).

More Fun with jar Tool What does that `-cvmf` mean? What else can you do with *jar* tool? The main purpose of the *jar* tool is to create archive files based on the ZIP format. It's a handy way to deliver an application, especially when it contains, for example, 4,234 classes. You *could* deliver all 4,243 to your end-users, but just *one* file is a little simpler. JAR files are also handy when you're shipping something over a network, where *one* fat file will ship faster than a ton of individual ones.

You can do all the things you'd expect to do with an archive format: make one, extract files from one, and look inside one (to display, but not extract, the contents). Table 18-1 lists examples of the basic *jar* tool commands. Assume that the JAR file name will be `MyJar.jar` and the class file we'll put in the JAR is `MyClass.class.`

TABLE 18-1 Basic *jar* Tool Commands

To create a JAR	`jar -cf  MyJar.jar MyClass.class`
To extract the contents of a JAR	`jar -xf MyJar.jar`
To view the contents of a JAR	`jar -tf MyJar.jar`
To create a JAR with your own manifest	`jar -cmf myManifest.mf MyJar.jar MyClass.class`
To create a JAR that contains all files within a directory	`jar -cf MyJar.jar suncertify/db`

The key *jar* tool switches are described here:

c	Create a new JAR file
v	Verbose—print out messages while JARring
t	List the contents of a file (in other words, display a *table*)
f	Stands for archive file name
m	Manifest—you are specifying your own text file whose contents should be put into the *real* manifest file.

Running the Executable JAR

Now that you've made your JAR file and it's got your complete package structure—with your main class in its appropriate directory (matching the package hierarchy)—and you've got a manifest inside that says *which* of your classes in the JAR has *the* `main()` method, it's a snap to run it from the command-line:

```
java -jar runme.jar
```

This works only if…

- Inside `runme.jar` there's a manifest.
- Inside the manifest, it specifies which class has a `main()` method, using the Main-Class entry:
  `Main-Class: suncertify.db.ExamProject`
- The manifest Main-Class entry specifies the fully-qualified name of the class (in other words, you must include the package structure).

- The ExamProject class is located inside the directory structure matching the package hierarchy. In other words, immediately inside the JAR there is a `suncertify` directory that contains a `db` directory, and the `db` directory contains the `ExamProject.class` file.

What About Command-Line Arguments? Your project's main application will almost certainly need command-line arguments. They might be used to pass host names and/or port numbers, or a variety of other configuration information. Most likely, though, your instructions will restrict you to a very small list of command-line arguments. For example, you might be instructed to use something like the following:

- Use `server` to have the application launch the network server.
- Use `client` to have the application launch the GUI client.
- The default—no argument—tells the application to run in *standalone* mode. (*Standalone* mode means your GUI must access the database *without* going through the network server. In other words, without RMI or sockets.)

Passing arguments to the command-line of your main class looks like this:

```
java -jar runme.jar  server
```

So anything you type *after* the JAR name is a command-line argument that gets passed straight on through to the `main()` method of your main class. In the example above, the ExamProject `main()` method would get the string server at `args[0]`.

Running javadoc on Your Package Chapter 17 describes the details of *javadoc*, but we thought a little reminder about running *javadoc* on your package might be helpful here. For this example, assume that

- Your class files are in the `suncertify/db` directory.
- Your working directory is one level *above* the start of the package (in other words, one level above the `suncertify directory)`.
- You have created a directory called `javadocs` and placed it within the docs directory.

To create javadocs for all classes in the `suncertify.db` package and have them placed in the `docs/javadocs` directory (where they'll need to be in our sample packaging instructions), run the following command from the directory one level *above* both `docs` and `suncertify`:

```
javadoc -d ./docs/javadocs suncertify.db
```

The Follow-Up Essay

Immediately after submitting your project (which means uploading it to the location you're given in your instructions), you should schedule your follow-up essay exam! We can't emphasize strongly enough that you should take your follow-up exam at the earliest possible moment. The fresher the project is in your mind, the easier it will be. You take the follow-up exam just as you did the Programmer's exam—in a Prometric Testing Center, where you answer questions in a computer application.

Rather than multiple-choice questions, however, your follow-up consists solely of essay questions. One of the main purposes of the follow-up is to see that indeed *you* are the one who wrote the application. So, you'll get questions related to the design and implementation decisions you had to make along the way. Once you've completed your essay exam, the results of the essay along with your submitted project are sent to an assessor and the fun begins. Well, for the assessor anyway. You, on the other hand, have to sit there pulling your hair out until you get your results. *Which could take as long as six weeks!* Unlike the Programmer's exam, where the results show up instantly, before you even leave the testing center, the Developer exam is marked by a real, *live* (which is a good thing, we're told…dead assessors have too many issues) human being. They'll use your essay questions to understand what's really happening.

Follow-Up Essay Questions

You never know what kind of questions you're going to get. Be prepared to describe *anything* that you might have had to consider in your design or implementation. The following questions don't necessarily reflect the actual exam questions, but they do suggest what you need to think about when preparing. We recommend that you relax about the follow-up! You've done the crucial work by designing and building your application, and assuming you—and not your cousin Chad who owes you—did the project yourself, you shouldn't have any trouble with the follow-up. Remember, there is NO ONE RIGHT ANSWER. There is, however, a *wrong* answer: the answer that contradicts the design, structure, and implementation of your application. As long as you're consistent, however, you should have nothing to fear from this part of the exam. The following questions can help you prepare for the follow-up:

- How did you decide which GUI components to use in your client application?
- Describe your design tradeoffs, and final decision, for choosing RMI or Sockets.
- Describe your event-handling mechanism and why you chose to do it that way.
- Describe your overall locking scheme and why you chose that approach.
- What design patterns did you use on this application?
- What aspects of your design allow for future modifications?
- Describe your threading and synchronization decisions.

Once you've submitted your follow-up assignment and taken your follow-up essay exam, take a deep breath, relax, and consider a refreshing alcoholic beverage. Or three. You're done! Now all that's left is the staggeringly torturous, endless, agonizingly slow, *wait* for the results. And after that, and after you've recovered from the "no more certifications for me no matter what you hear me say" phase, you can start planning your *next* certification effort.

In the meantime, don't count your chickens until you've reviewed the following key points summary. And *then* you're done. (Well, except for re-reading the threads chapter.) (Not to mention learning the exotic subtleties of the JTable.)

Key Points Summary

- Follow your project submission instructions *perfectly.*
- Your project will probably require you to package up all files in a single JAR.
- Your application should be (your instructions will confirm this) an executable JAR.
- You will need to submit both source code and class files.

- Class files will be in the executable JAR, while source code will be in a specific directory, probably named "code".
- Your instructions will give you the exact layout of the top-level project JAR including where to place the documentation, code, design choices document, and user guide.
- To create an executable JAR, you need a class with a `main() method` and a manifest with a Main-Class entry.
- The Main-Class entry indicates which class in the JAR has the `main()` method that should be executed, and it looks like the following: `Main-Class: suncertify.db.ExamProject`
- The *jar* tool will create a manifest; you will create a text file and specify to the *jar* tool that it should take your text file and incorporate its contents into the *real* manifest.
- The real manifest is `META-INF/MANIFEST.MF.`
- To run the *jar* tool on all the classes in a package, place them in a new JAR named `runme.jar` and include your own manifest file information, using the following command:

  ```
  jar -cvmf Manifest.MF runme.jar suncertify/db
  ```
- To run the executable JAR, use the following command-line:

  ```
  java -jar runme.jar
  ```
- You can also pass command-line arguments to the `main()` method in the main class by placing them after the JAR name as follows:

  ```
  java -jar runme.jar  standalone
  ```
- The purpose of the follow-up essay is designed to verify that *you* are the one who actually developed the assignment application.
- Schedule your follow-up essay exam *as soon as you upload/submit your assignment.*
- The fresher the project is in your mind, the better for taking the follow-up essay exam.
- For the follow-up essay, be prepared to describe and defend choices you made while designing and implementing your application.

- It's completely natural to (on many occasions) *regret* the decision to try for this certification in the first place. The feeling will pass. When it does, *get busy*, it'll be back.
- We wish you luck, success, caffeine, and plenty of sleep.
- When you've received your certification, tell us! Drop a success-story to certified@wickedlysmart.com.