Java Platform, Standard Edition

Packaging Tool User's Guide

Release 25
G29150-01
September 2025

ORACLE

Java Platform, Standard Edition Packaging Tool User's Guide, Release 25

G29150-01

Contents

Preface

This guide provides information about using `jpackage`, the packaging tool provided with the JDK for generating installable packages for self-contained Java applications.

Audience

This guide is intended for developers interested in creating self-contained Java applications that provide native packaging formats, which give the end user a natural installation experience.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See JDK 25 Documentation.

Conventions

The following conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Packaging Overview

The packaging tool `jpackage` enables you to generate installable packages for modular and non-modular Java applications. Platform-specific packages for Linux, macOS and Windows provide your users with a familiar way to install and launch your applications.

The simplest form of packaging takes a pre-built Java application as input and generates an installable package in a platform-dependent default format. The packaging tool generates a runtime for your application using the `jlink` command.

For applications that require more advanced capabilities, command line options are available for features such as the following:

- Provide a custom icon
- Install the application in a specific location
- Specify JVM options and application arguments to be used when launching the application
- Set file associations to launch the application when an associated file type is opened
- Launch the application from a platform-specific menu group
- Set up multiple launchers for the application
- Sign the bundle (macOS only)

For a description of `jpackage` and its options, see The jpackage Command in *Java Development Kit Tool Specifications*.

Topics:

Packaging Pre-Reqs

Application packages must be built on the target platform. The system used for packaging must contain the application, a JDK, and software needed by the packaging tool.

To package your application for multiple platforms, you must run the packaging tool on each platform. If you want more than one format for a platform, you must run the tool once for each format.

The following platforms and formats are supported with the required software:

- Linux: `deb`, `rpm`:
  - For Red Hat Linux, the rpm-build package is required.
  - For Ubuntu Linux, the fakeroot package is required.
- macOS: `pkg`, `app` in a `dmg`

Xcode command line tools are required when the `--mac-sign` option is used to request that the package be signed, and when the `--icon` option is used to customize the DMG image.

- Windows: `exe`, `msi`

  WiX 3.0 or later is required.

Application Preparation

To package your application, you must first build it and create the necessary JAR or module files. Resources needed by your application must also be available on the system used for packaging.

The following application-related information and resources are used for packaging:

- JAR or module files for the application
- Application metadata, for example, name, version, description, copyright, license file
- Installation options, for example, shortcut, menu group, additional launchers, file associations
- Launch options, for example, application arguments, JVM options

As part of the packaging process, an application image based on the files in the input directory is created. This image is described in Generated Application Image. To test your application before creating an installable package, use the `--type app-image` option to create only the application image.

Generated Application Image

The packaging tool creates an application image based on the input to the tool.

The following example shows the application image created for a simple Hello World application for each platform. Files that are considered implementation details are subject to change and are not shown.

- Linux:

```
myapp/
  bin/              // Application launchers
    HelloWorld
  lib/
    app/
      HelloWorld.cfg     // Configuration info, created by jpackage
      HelloWorld.jar     // JAR file, copied from the --input directory
    runtime/             // Java runtime image
```

- macOS:

```
HelloWorld.app/
  Contents/
    Info.plist
    MacOS/               // Application launchers
      HelloWorld
    Resources/           // Icons, etc.
    app/
      HelloWorld.cfg     // Configuration info, created by jpackage
```

```
        HelloWorld.jar     // JAR file, copied from the --input directory
      runtime/             // Java runtime image
```

- Windows:

```
HelloWorld/
  HelloWorld.exe       // Application launchers
  app/
    HelloWorld.cfg     // Configuration info, created by jpackage
    HelloWorld.jar     // JAR file, copied from the --input directory
  runtime/             // Java runtime image
```

The application image generated by the tool works for most applications. However, you can make changes before packaging the image for distribution, if needed.

Java Runtime Requirements

To eliminate the need for users to install a Java runtime, one is packaged with your applications. The packaging tool generates a runtime image based on the packages or modules that your application needs.

If no Java runtime image is passed to the packaging tool, then `jpackage` uses the `jlink` tool to create a runtime for the application. Runtime images created by the packaging tool do not contain debug symbols, the usual JDK commands, man pages, the `src.zip` file, or service bindings (see the `Configuration` class).

For non-modular applications composed of JAR files, the generated runtime image contains the same set of JDK modules that is provided to class-path applications in the unnamed module by the regular `java` launcher. The following example creates a self-contained Java application composed of one JAR file:

```
jpackage --name DynamicTreeDemo \
         --input . --main-jar DynamicTreeDemo.jar \
         --module-path $JAVA_HOME/jmods
```

For modular applications composed of modular JAR files and JMOD files, the generated runtime image contains the application's main module and the transitive closure of all of its dependencies. The following example creates a self-contained Java application composed of one module:

```
jpackage --name Hello --module-path mods \
         --module com.greetings/com.greetings.Main
```

To add additional modules, use the `--add-modules` option. The following examples add the `java.logging` module:

```
jpackage --name DynamicTreeDemo \
         --input . --main-jar DynamicTreeDemo.jar \
```

```
          --module-path $JAVA_HOME/jmods \
          --add-modules java.logging


jpackage --name Hello --module-path "mods:$JAVA_HOME/jmods" \
          --module com.greetings/com.greetings.Main \
          --add-modules java.logging
```

In JDK 25 and later, the generated runtime image doesn't include service bindings. You can add them with the `--jlink-options` option and passing it the `--bind-services` `jlink` option:

```
jpackage --name DynamicTreeDemo \
          --input . --main-jar DynamicTreeDemo.jar \
          --jlink-options --bind-services


jpackage --name Hello --module-path mods \
          --module com.greetings/com.greetings.Main \
          --jlink-options --bind-services
```

Note

- If you don't specify the `--jlink-options` option, then, by default, the `jpackage` tool adds these `jlink` options: `--strip-native-commands`, `--strip-debug`, `--no-man-pages`, and `--no-header-files`.
- In JDK 25 and later, `jpackage` no longer includes service bindings in the runtime image that it creates. Prior to JDK 25, `jpackage` would include them. As a result, the generated runtime images produced by `jpackage` in JDK 25 and later might not include the same set of modules as it did in prior JDK releases.

  To include the same set of modules in the generated runtime image as in previous JDK releases, use the `--jlink-options` option and pass it the `--bind-services` `jlink` option in addition to the default `jlink` options that `jpackage` uses:

  ```
  jpackage [...] --jlink-options --strip-native-commands --strip-
  debug \
                 --no-man-pages --no-header-files --bind-services
  ```

The runtime image generated by the tool works for most applications. However, you can create a custom runtime to package with your application, if needed.

2
Basic Packaging

If your application doesn't require customizations or support for features such as multiple launchers or file associations, then only a couple of options are needed for packaging.

The simplest form of packaging requires the location of the application to package and the name of the JAR or module that contains the main class.

- The following example packages a non-modular application:

  ```
  jpackage --input app-directory --main-jar jar-file [--main-class main-class]
  ```

  app-directory is the name of the directory that contains the files for your application. The path can be absolute or relative to the current directory. *jar-file* is the name of the JAR file that contains the main class for the application. *main-class* is the name of the main class and is only required if the main class is not identified in the `MANIFEST.MF` file. Tool and platform defaults are used as needed to complete the package.
- The following example packages a modular application:

  ```
  jpackage --module-path module-path --module main-module[/class]
  ```

  module-path is the path to either a directory of modules or to a modular JAR file. The path can be absolute or relative to the current directory. For more than one path, separate the paths with a colon (:) on Linux and macOS or a semi-colon (;) on Windows, or use multiple instances of the `--module-path` option. *main-module/class* is the name of the module that contains the main class and the name of the main class for the application. The name of the main class is only required if the main module does not identify the main class. Tool and platform defaults are used as needed to complete the package.

Topics:

Defaults for Options Not Specified

Options are available to control the name of the application, type of package created, installation location, and other characteristics of the package. If an option is not provided, a default values is used.

The following defaults apply to options that are not specified when running `jpackage`:

- The package type is platform-dependent:
  - On Linux, the default is `deb` for Debian Linux and `rpm` for other versions of Linux.

  - On macOS, the default is `dmg`.
  - On Windows, the default is `exe`.

  To generate a different type of package, use the `--type` option.

- The generated package is written to the current working directory. To write the package to a different location, use the `--dest` option.
- The name of the package is generated from the name of the application and the application version. If no application name is provided, the name of the main JAR or module is used, followed by the version, which defaults to 1.0, for example `HelloWorld-1.0.exe`. To change the name of the application, use the `--name` option. To change the version, use the `--app-version` option.
- The Java runtime is generated during the packaging process using the `jlink` command. The `--add-modules` and `--jlink-options` options can be used to add items to the runtime as part of the packaging process. To package a custom runtime, use the `--runtime-image` option.
- The installation directory is platform-specific:
  - On Linux, the default is `/opt/`*`application-name`*
  - On macOS, the default is `/Applications/`*`application-name`*
  - On Windows, the default is `c:\Program Files\`*`application-name`*; if the `--win-per-user-install` option is used, the default is `C:\Users\`*`user-name`*`\AppData\Local\`*`application-name`*

  The name of the application directory defaults to the name of the application. To give the directory a different name, use the `--install-dir` option.
- The name of the application launcher defaults to the name of the application. If your application has more than one launcher, use the `--add-launcher` option to identify them.
- No default command line arguments or Java runtime options are passed to the application when it is started. The user can pass application arguments from the command line when launching the application, but not Java runtime options.
- A default icon for the application is used. For a different icon, use the `--icon` option.
- For Linux, the name of the package defaults to the application name. To give the package a different name, use the `--linux-package-name` option.
- For macOS:
  - The application identifier defaults to the main class name. To use a different identifier, use the `--mac-package-identifier` option.
  - The name of the application shown in the menu bar defaults to the main class name of the application. To use a different name, use the `--mac-package-name` option.

Package a Non-Modular Application

A non-modular application package can be packaged by providing just the location of the files to package and the name of the main JAR file. Defaults are used for other options that describe the package and the application.

The following command when run on a Windows system packages the non-modular application in the `mySamples\hwapp` directory with the main class in the `HelloWorld.jar` file.

```
jpackage --input mySamples\hwapp --main-jar HelloWorld.jar
```

Because no other options are used, the following defaults are applied:

- The default type of package generated is `exe`.
- The name of the package generated is `HelloWorld-1.0.exe`.
- The package is written to the current directory.
- The runtime packaged with the application is generated as part of the packaging process.
- The application is installed in the `c:\Program Files\HelloWorld` directory.
- The name of the launcher is `HelloWorld.exe`.
- The default icon is used for the application.
- No shortcut is created, and the application is not added to any menu. The user must go to the directory in which the application is installed to run it.
- No default arguments or Java runtime options are passed to the application when it is started.

Package a Modular Application

A modular application package can be packaged by providing just the location of the modules to package and the name of the main module. Defaults are used for other options that describe the package and the application.

The following command when run on a Debian Linux system packages the modular application in the `myModApps` directory with the main class in the `modhw/modhw.HelloWorldMod` module

```
jpackage --module-path myModApps --module modhw/modhw.HelloWorldMod
```

Because no other options are used, the following defaults are applied:

- The default type of package generated is `deb` for Debian systems
- The name of the package generated is `HelloWorldMod-1.0.deb`.
- The package is written to the current directory.
- The runtime packaged with the application is generated as part of the packaging process.
- The application is installed the `/opt/HelloWorldMod` directory.
- The name of the launcher is `HelloWorldMod`.
- The default icon is used for the application.
- No shortcut is created, and the application is not added to any menu. The user must go to the directory in which the application is installed to run it.
- No default arguments or Java runtime options are passed to the application when it is started.

Identify Your Application with Package Metadata

As you create the package, you might want to provide information about the application, such as a description, the vendor name, or perhaps a copyright statement.

To add information about your application to the package, use the relevant `jpackage` options to set the package metadata. The following examples are for a Windows system.

- Set the name of the application.

  Use the `--name` option to give the application the name that you want users to see. If no name is provided, it defaults to the name of the main JAR file or module.

  The following command creates a package for the Dynamic Tree application named `DynamicTreeDemo-1.0.exe`:

  ```
  jpackage --name DynamicTreeDemo --input myApps \
     --main-jar DynamicTree.jar
  ```

- Set the application version.

  Use the `--app-version` option to identify the version of your application. If no application version is specified, the version defaults to 1.0.

  The following command customizes the version part of the package name and creates the package `DynamicTreeDemo-2.0.exe`:

  ```
  jpackage --name DynamicTreeDemo --app-version 2.0 \
     --input myApps --main-jar DynamicTree.jar
  ```

- Describe the application.

  Use the `--description` option to include a brief description of your application. No default description is provided.

  The following command describes the Dynamic Tree application to users; note that quotes are required if the description includes spaces:

  ```
  jpackage --dest packages --name DynamicTreeDemo \
     --app-version 2.0 --input myApps --main-jar DynamicTree.jar \
     --description "Demo application for testing functionality"
  ```

- Set the vendor for the application.

  Use the `--vendor` option to identify yourself or your company as the creator of your application. No default vendor is provided.

  The following command identifies the vendor of the Dynamic Tree application as Small, Inc; note that quotes are required if the vendor name includes spaces:

  ```
  jpackage --dest packages --name DynamicTreeDemo \
     --app-version 2.0 --input myApps --main-jar DynamicTree.jar \
     --description "Demo application for testing functionality" \
     --vendor "Small, Inc"
  ```

- Set the copyright for the application.

  Use the `--copyright` option to provide a copyright for your application. No default copyright is provided.

  The following command provides an example of a copyright statement for the Dynamic Tree application; note that quotes are required if the copyright includes spaces:

  ```
  jpackage --dest packages --name DynamicTreeDemo \
     --app-version 2.0 --input myApps --main-jar DynamicTree.jar \
     --description "Demo application for testing functionality" \
     --vendor "Small, Inc" --copyright "Copyright 2020, All rights reserved"
  ```

3

Support Application Requirements

The packaging tool provides support for application requirements such as default arguments, JVM options, file associations, multiple launchers, and signing.

Topics:

Set Default Command-Line Arguments

If your application accepts command-line arguments, use the `--arguments` option to define default values. Users can override these values when they start the application.

If you package your application with default command-line arguments, these values are passed to the main class when the user starts your application without providing arguments. The `[ArgOptions]` section of the *`app-name`*`.cfg` file in the `/app` directory of the application image generated by `jpackage` shows any default arguments that are defined. You can check this file to ensure that the values are defined correctly.

The following examples show some of the ways to set up default arguments:

- Set the default value for a single argument.

  The following command defines the value for a single argument for the MyApp application.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --arguments arg1
  ```

- Set the default value for more than one argument.

  Use a space to separate arguments and enclose the entire string in quotes, or use multiple instances of the `--arguments` option. The following commands show alternative ways to define three default command-line arguments for the MyApp application.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --arguments "arg1 arg2 arg3"

  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --arguments arg1 --arguments "arg2 arg3"

  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --arguments arg1 --arguments arg2 --arguments arg3
  ```

- Set a default value that contains spaces.

If an argument contains a space, two sets of quotes are needed to ensure that `jpackage` treats the spaces as part of the value and not as delimiters between values. Enclose the argument in single quotes, or double quotes preceded by the escape character, then enclose the quoted string in quotes. The following commands show alternative ways to define two arguments that contain spaces.

```
jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
   --arguments "\"String 1\" \"String 2\""

jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
   --arguments "\"String 1\"" --arguments "\"String 2\""

jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
   --arguments "'String 1'" --arguments "'String 2'"
```

Set JVM Options

If you want options passed to the JVM when your application is started, use the `--java-options` option when you package your application. Users can't provide JVM options to the application.

To set up the JVM as needed to run your application, define the JVM options to pass when a user starts your application. Use the `$APPDIR` macro to reference resources included with the application. The resource file must be in the input directory when the application is packaged.

The [JavaOptions] section of the *`app-name`*`.cfg` file in the `/app` directory of the application image generated by `jpackage` shows any default arguments that are defined. You can check this file to ensure that the values are defined correctly.

The following examples show some of the ways to pass JVM options to your application:

- Set a single JVM option.

  The following command sets the initial size of the heap for the MyApp application to 2 megabytes.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --java-options Xms2m
  ```

- Set more than one JVM option.

  To provide more than one JVM option, use a space to separate arguments and enclose the entire string in quotes, or use multiple instances of the `--jvm-options` option. The following commands show alternate ways to set the initial size and the maximum size for the heap.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --java-options "Xms2m Xmx10m"

  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --java-options Xms2m  --java-options Xmx10m
  ```

- Set a JVM option that contains a space.

  If a JVM option contains a space, two sets of quotes are needed to ensure that `jpackage` treats the spaces as part of the option and not as delimiters between options. Enclose the argument in single quotes, or double quotes preceded by the escape character, then

enclose the quoted string in quotes. The following commands show alternate ways to define an option that contain spaces.

```
jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
   --java-options "\"-DAppOption=text string\""

jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
   --java-options "'-DAppOption=text string'"
```

- Set a JVM option that contains quotes.

  If a JVM option contains quotes, escape characters must be used for the quotes. The following command passes the JVM option `-XX:OnError="userdump.exe %p"` to `jpackage`.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --java-options "-XX:OnError=\"\\\"userdump.exe %p\\\"\""
  ```

- Use the $APPDIR macro with a JVM option.

  To use the image `myAppSplash.jpg` from the application directory as the splash screen for your application, use the `$APPDIR` macro as shown in the following example. The image file must be in the input directory when the application is packaged. Note that in some shells the dollar sign needs to be escaped, for example, `\$APPDIR`.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --java-options "-splash:\$APPDIR/myAppSplash.jpg"
  ```

Set Class and Module Paths

By default, the `jpackage` tool generates a default class path that contains the path to each JAR file that's specified in the `--input` option. However, a class path that you specify with the `-cp`, `-classpath`, or `-Djava.class.path` option through the `--java-options` option overrides the default class path.

If you're using `--java-options` to specify the class path, then ensure that you include the path to each of your input JAR files in it. For example, if your application contains only one JAR file, `myapp.jar` but you want to include the classes in the `classes` subdirectory in the class path, then add the following to the `jpackage` command:

```
--java-options "-cp \$APPDIR/myapp.jar:\$APPDIR/classes"
```

For a modular application, the default module path that `jpackage` generates is `$APPDIR/mods`. However, if you specify a module path with the `--module-path` option through `--java-options` , then your module path is appended after the default module path; it doesn't replace the default one.

Set File Associations

If you want your application to be started when a user opens a specific type of file, use the `--file-associations` option when you package your application.

To have your application started when a user opens a file that your application can handle, define the file associations that you want created when the application is installed. Associations are defined in properties files that are passed to `jpackage`. For each association,

a separate file and a separate instance of the `--file-associations` option is required. The following properties define an association, which must include either `mime-type` or `extension`:

- `mime-type` - MIME type for files that your application can process.
- `extension` - File extension for files that your application can process.
- `icon` - Icon to use for the type of files that your application can process. The icon must be in the input directory when the application is packaged. If no icon is specified, the default icon is used.
- `description` - Short description of the association.

To set up file associations, first create the properties files. The following two files set up an association for JavaScript files and for Groovy files.

`FAjavascript.properties:`

```
mime-type=text/javascript
extension=js
description=JavaScript Source
```

`FAgroovy.properties:`

```
mime-type=text/x-groovy
extension=groovy
description=Groovy Source
```

The following command packages the application FADemo and sets up file associations using the properties files just created. When a user opens a `.js` or `.groovy` file, FADemo is started.

```
jpackage --name FADemo --input FADemo \
   --main-jar ScriptRunnerApplication.jar \
   --file-associations FAjavascript.properties \
   --file-associations FAgroovy.properties
```

Add Launchers

If you have more than one way start your application, use the `--add-launcher` option to describe the additional launchers that you want created.

You might want an additional launcher if your application has different default values for arguments or can run with or without the Windows console, or if you package multiple apps together to share a runtime. The format for the option is `--add-launcher` *`launcher-name=properties-file`*, where *launcher-name* is the named used for the additional launcher. Use quotes if the name contains spaces.

The launchers are defined in properties files that are passed to `jpackage`. For each launcher, a separate file and a separate instance of the `--add-launcher` option is required. The following properties define a launcher, at least one option must be set:

- `module` - Name of the module that contains the main class for the launcher. If the main module does not identify the main class, include it in the format `module=`*`main-module/class`*.
- `main-jar` - Name of the JAR file that contains the main class for the launcher.
- `main-class` - Name of the main class.

- `arguments` - Default arguments, separated by spaces. If an argument contains spaces, enclose the argument in quotes, for example, `arguments=arg1 "arg 2" arg3`
- `app-version` - Version number.
- `java-options` - Options to pass to the JVM, separated by spaces. If an argument contains spaces, enclose the argument in quotes.
- `icon` - Icon used for the additional launcher
- `win-console` - Set to `true` to start the console with the application.

To define additional launchers, first create the properties files. The following examples show some of the ways to set up a launcher:

- Add a launcher with different application arguments.

  Create the following properties files that define different default arguments to use when the application is launched. The first file defines 3 arguments to pass. The second file defines two arguments to pass.

  `MLAppArgs1.properties:`

  ```
  arguments=arg1 arg2 arg3
  ```

  `MLAppArgs2.properties:`

  ```
  arguments="String 1" "String 2"
  ```

  The following command packages the application MyApp with two additional launchers using the properties files just created.

  ```
  jpackage --name MyApp --input samples/myapp --main-jar MyApp.jar \
     --add-launcher MyApp1=MLAppArgs1.properties \
     --add-launcher MyApp2=MLAppArgs2.properties
  ```

- Add a launcher to start the Windows console.

  To provide the user with the option of running your application with or without the console, create the following properties file that defines a launcher that uses the Windows console.

  `MLConsole.properties:`

  ```
  win-console=true
  ```

  The following command packages the HelloWorld application with an additional launcher that runs the application with the Windows console.

  ```
  jpackage --name HelloWorld --input helloworld \
     --main-jar HelloWorld.jar \
     --add-launcher HWConsole=MLConsole.properties
  ```

- Add a launcher for a second entry point.

  When more than one application is included in the same package, each application can be started independently by adding additional launchers. If the FADemo and the Dynamic Tree applications are packaged together and the main launcher is for the FADemo

application, create the following properties file to define an additional launcher for the Dynamic Tree application.

MLDynamicTree.properties

```
main-jar=DynamicTree.jar
main-class=webstartComponentArch.DynamicTreePanel
icon=DTDemo.ico
```

The following command packages the two applications together and sets up the additional launcher using the properties file just created.

```
jpackage --name MLDemo --input MLDemo \
   --main-jar ScriptRunnerApplication.jar \
   --add-launcher "Dynamic Tree"=MLDynamicTree.properties
```

Sign the Application Package (macOS)

For an application that runs on macOS, use the `--mac-sign` and supporting options when you package your application. A disk image (`.dmg`) or package (`.pkg`) that contains a signed application image (`.app`) can be notarized.

The required `jpackage` options depend on whether or not you want to distribute your application though the Mac App Store.

Required Certificates

If you want to distribute your application outside the Mac App Store, then you'll need the certificates "Developer ID Application: *<user or team name>*" and "Developer ID Installer: *<user or team name>*".

If you want to deploy your application through the Mac App Store, then you'll need the certificates "3rd Party Mac Developer Application: *<user or team name>*" and "3rd Party Mac Developer Installer: *<user or team name>*".

Options for Signing macOS Application Package

To sign a macOS application package, include the following `jpackage` options:

- `--mac-sign`: Requests that the bundle be signed for macOS.
- `--mac-signing-key-user-name` *`user_or_team_name`*: The key user or team name, which is the name portion in Apple signing identities' names.

In addition, you may require the following options

- `--mac-package-signing-prefix` *`prefix`*: When signing the application bundle, this value is prefixed to all components that need to be signed that don't have an existing bundle identifier. If you don't specify this option, then the prefix is the (unqualified) main class name followed by a period (`.`).
- `--mac-signing-keychain` *`keychain_name`*: If a keychain other than the standard keychain is used, then specify the name of the keychain as show in the Keychain Access app. The name should end in `.keychain`.
- `--type` *`type`*: If you want to create an application image (`.app`), specify `app-image`; if you want to create a package (`.pkg`), specify `pkg`. If you don't specify this option, then this option creates a disk image (`.dmg`).

- `--mac-entitlements` *`path`*: Path to the file containing entitlements to use when signing executables and libraries in the bundle.

  If you don't specify the `--mac-entitlements` option nor the `--mac-app-store` option, then `jpackage` uses the entitlements file `default.plist`, which is a built-in resource (see Resources Used in Packaging). It contains entitlements that enable your signed application to run the JDK.

The following command generates a disk image (`.dmg`) containing an application image signed with the "Developer ID Application: developer.example.com" certificate. The disk image is generated with the prefix `com.example.developer.OurApp.` and the team name developer.example.com.

```
jpackage --name DynamicTreeDemo --input myApps --main-jar DynamicTree.jar \
   --mac-sign --mac-package-signing-prefix com.example.developer.OurApp. \
   --mac-signing-key-user-name "developer.example.com"
```

The following command generates a package (`.pkg`) containing an application image signed with the "Developer ID Installer: developer.example.com" certificate. The package is generated with the prefix `com.example.developer.OurApp.` and the team name developer.example.com.

```
jpackage --type pkg --name DynamicTreeDemo --input myApps \
   --main-jar DynamicTree.jar --mac-sign --mac-package-signing-prefix
com.example.developer.OurApp. \
   --mac-signing-key-user-name "developer.example.com"
```

Options for Signing Application Package for Mac App Store

To sign an application package for the Mac App Store, also include the following `jpackage` options:

- `--mac-app-store`: Indicates that the `jpackage` output is intended for the Mac App Store.
- `--mac-entitlements` *`path`*: Path to file containing entitlements to use when signing executables and libraries in the bundle. This file should enable the App Sandbox Entitlement, which restricts your application to system resources and user data. It's required for applications distributed through the Mac App Store.

  If you don't specify the `--mac-entitlements` option but specify the `--mac-app-store` option, then `jpackage` uses the entitlements file `sandbox.plist`, which is a built-in resource (see Resources Used in Packaging). It contains `<key>com.apple.security.app-sandbox</key><true/>`, which enables the App Sandbox Entitlement.
- `--mac-app-category` *`category`*: Specifies the category that best describes your application package for the Mac App Store. The `jpackage` tool sets the value of LSApplicationCategoryType to the value of this option in your application's `.plist` file. The default value of this option is `utilities`. See LSApplicationCategoryType in *Apple Developer Documentation* for a list of valid categories.

4

Manage the Installation of Your Application

You have some control over how your application is installed and launched on the user's system. Using options provided by the packaging tool, you can specify such things as the license to be accepted, where to install the application, and if a console is needed.

Topics:

Include a License

If you have terms and conditions that you want users to accept to install your application on Windows or macOS, use the `--license-file` option when you package your application.

If the directory that contains your application also includes a license file, that file is installed on the user's machine with the application. If you want to require the user to accept the license before installing on Windows or macOS, use the `--license-file` option. Be aware that if you provide a license file that is not in the application directory, the user is shown the license when installing, but the file is not installed with the application. Also, for silent and other types of installs, the license file is not shown.

The following command adds the license file `myApps/myLicense.txt` to the package for the Dynamic Tree application.

```
jpackage --type exe --name DynamicTreeDemo --input myApps \
   --main-jar DynamicTree.jar --license-file myApps/myLicense.txt
```

Create a Shortcut

To have a shortcut created when users install your application, use the `--linux-shortcut` or `--win-shortcut` option when you package your application. To show a custom icon for your application, use the `--icon` option.

Shortcuts are supported for Linux and Windows platforms. If you don't provide an icon, a default icon is used. If you provide a custom icon on Linux, a shortcut is automatically created and the `--linux-shortcut` option is not needed. Custom icons must be in a format that meets the requirements of the platform.

The following command creates a shortcut with the default icon for the Dynamic Tree application when it is installed on Linux.

```
jpackage --name DynamicTreeDemo --input myApps --main-jar DynamicTree.jar \
   --linux-shortcut
```

The following command creates a desktop shortcut with a custom icon for the Dynamic Tree application when it is installed on Windows.

```
jpackage --name DynamicTreeDemo --input myApps --main-jar DynamicTree.jar \
   --icon DTDemo.ico --win-shortcut
```

Set the Installation Directory

If you want the name of the installation directory to be different than the name of the package, use the `--install-dir` option. On Windows you can let the user choose where to install your application by using the `--win-dir-chooser` option.

Your application is installed in the default platform-specific installation directory described in Defaults for Options Not Specified. The directory name for the application defaults to the package name, but this can be changed with the `--install-dir` option when you package the application.

On Windows, you also have the option to enable the user to choose the installation location. The dialog shown defaults to a directory with the package name.

The following command installs the Dynamic Tree application in `c:\Program Files\DTDemo` instead of `c:\Program Files\DynamicTreeDemo`.

```
jpackage --type exe --name DynamicTreeDemo --input myApps \
   --main-jar DynamicTree.jar --install-dir DTDemo
```

The following command lets the user choose the directory where the application is installed.

```
jpackage --type exe --name DynamicTreeDemo --input myApps \
   --main-jar DynamicTree.jar --win-dir-chooser
```

Add the Application to a Menu

To let users access your application from a menu, use the `--linux-menu-group` option, or the `--win-menu` and `--win-menu-group` options when you package your application.

On the Linux platform, if the `--linux-menu-group` option is not used, your application is added to the Unknown group in a menu specific to the window manager being used.

On the Windows platform, you can have your application added to the Start menu in the group of your choosing. If the group doesn't exist, it is created. If you don't provide a group name, the application is added to the Unknown group. The `--win-menu-group` option is only meaningful if the `--win-menu` option is used.

The following command adds the Dynamic Tree application to the Windows Start menu in the "Small, Inc" group. Quotes are needed only if the name includes spaces.

```
jpackage --type exe --name DynamicTreeDemo --input myApps \
   --main-jar DynamicTree.jar --win-menu --win-menu-group "Small, Inc"
```

On macOS, the application is shown in the menu bar. The name shown defaults to the name of the package. The following command uses the `--mac-package-name` option to show DTDemo in the menu bar.

```
jpackage --name DynamicTreeDemo --input myApps --main-jar DynamicTree.jar \
   --mac-package-name DTDemo
```

Launch in Console

If your application runs from the command line or requires console interaction, use the `--win-console` option to let Windows know to start the application in a console window.

The following command tells Windows to start the Hello World application with a console window.

```
jpackage --input mySamples\hwapp --main-jar HelloWorld.jar --win-console
```

5

Image and Runtime Modifications

The application image and Java runtime generated by the packaging tool work well for most applications. However, you can make changes to the image and runtime for any custom requirements that you might have, and then use the modified version when packaging your application.

Topics:

Application Image Modifications

If needed, you can modify the application image that the packaging tool creates and then package the modified image for distribution.

Possible reasons for modifying the image include: adding removing files, adding resources, or changing the runtime. If you need to modify the image, run the packaging tool twice as follows:

1. Create only the application image with the `--type app-image` option. For example:

   ```
   jpackage --type app-image --name HelloWorld --module-path myModApps \
       --module modhw/modhw.HelloWorldMod
   ```

   In this example, a directory named `HelloWorld` is created in the current directory. The `HelloWorld` directory contains the application image, which contains the modular application in the `myModApps` directory whose main class is in the `modhw/modhw.HelloWorldMod` module. An installable bundle is not created.

2. After you make the necessary changes to the application image, run the packaging tool again to create an installable bundle with the modified image. For example:

   ```
   jpackage --type msi --app-image HelloWorld --name HelloWorld
   ```

Note

- The `--name` option is required when packaging an application image.
- The `--runtime-image` option is not allowed with `--app-image`. You will get the following error:

  ```
  Error: Mutually exclusive options [--runtime-image] and [--app-
  image]
  ```

  If you want to use a different runtime, then specify it when you first run `jpackage` to create the application image. For example:

  ```
  jpackage --type app-image --name HelloWorld \
      --runtime-image myCustomJRE --module-path myModApps \
      --module modhw/modhw.HelloWorldMod
  ```

Java Runtime Modifications

When you want more control over the Java runtime that is packaged with your application, you can create a custom runtime.

To create a custom Java runtime image for your application, run `jlink` before you package your application. Then pass the image produced to the packaging tool using the `--runtime-image` option. Reasons you might want to use a custom runtime image:

- Have more control over the options that are used to create the runtime
- Package your application with a different version of Java than the version used to run `jpackage`
- Use the same runtime for more than one application.

For example, the following commands create a JDK 14 runtime that includes JavaFX 13 modules, and then package that runtime with an application:

```
jlink --output jdk-14+fx --module path javafx-jmods-13 \
   --add modules javafx.web,javafx.media,javafx.fxml,java.logging

jpackage --name myapp --input lib --main-jar myApp.jar \
   --runtime-image jdk-14+fx
```

If you are packaging an application that requires an earlier version of the Java runtime, use the `--runtime-image` option The following command packages the JDK 11 runtime with your application:

```
jpackage --name myapp --input lib --main-jar myApp.jar \
   --runtime-image jdk-11.0.5
```

If your application requires a custom runtime based on an earlier version of the JDK, use the earlier version to run `jlink` and create the runtime image. Then use current JDK to run

`jpackage` and pass it the custom runtime. The following commands create a custom runtime using JDK 11.0.5 and package it using JDK 14:

```
c:\Program Files\Java\jdk-11.0.5\bin\jlink output my-jdk11 \
   --add-modules java.desktop,java.datatransfer

c:\Program Files\Java\jdk-14\bin\jpackage --name myapp --input lib \
   --main-jar myApp.jar --runtime-image my-jdk11
```

6

Override jpackage Resources

Advanced customization of the package generated is possible by overriding resources used by `jpackage`, such as background images and template files for properties and scripts. The `--resource-dir` option is used to provide the overrides to the tool.

If the default resources that `jpackage` uses when packaging an application don't meet your needs, create a directory and add your customized files to it. If you override a file, your custom file must contain all of the properties that the default contains. Pass the path to the directory to `jpackage` using the `--resource-dir` option. The path can be absolute or relative to the current directory.

> **Note**
>
> Resources such as icons, application version, application description, copyright, and others can be overridden from the command line. Use of the command line options is recommended when available.

The topics that follow describe the resources that you can override and explain how you can find out what the defaults are.

Topics:

Resources Used in Packaging

The packaging tool has default templates and other resources that it uses when it generates the package for your application.

The resources vary by platform and are described in the following sections. In most cases, resources overridden with command line options take precedence over resources in the resource directory. To override resources that can't be overridden from the command line, add your customized files to the resource directory that you pass to `jpackage`. Use the `--verbose` option described in View Resources to verify the name of the override file for each resource.

Linux (all versions)

- Icon file, *`launcher`*`.png`, for the main launcher and any additional launchers. Each launcher can have a separate icon. The file name must match the name of the application or the name of a launcher. If an icon file is not provided for a launcher, the default icon is used.
- Desktop shortcut file, *`launcher`*`.desktop`, for the main launcher and any additional launchers The file name must match the name of the application or the name of a launcher.

Linux DEB

- Control template, `control`. File that contains information about the application.
- Pre-installation script, `preinst`. Script that is run before the application is installed.
- Pre-removal script, `prerm`. Script that is run before the application is uninstalled.
- Post-installation script, `postinst`. Script that is run after installation completes.
- Post-removal script, `postrm`. Script that is run after the application is uninstalled.
- Copyright file, `copyright`. File that contains copyright and license information.

Linux RPM

- Specification for packaging, *`package-name`*`.spec`. Instructions for packaging the application.

macOS (all formats)

- Icon file, *`launcher`*`.icns`, for the main launcher and any additional launchers. More than one file can be provided. The file name must match the name of the application or the name of a launcher. If an icon file is not provided for a launcher, the default icon is used.
- Runtime properties list, `Runtime-Info.plist`.
- Information properties list, `Info.plist`.
- Default entitlements file, *`application-name`*`.entitlements`.
- Post-image script, *`application-name`*`-post-image.sh`. Custom script that is executed after the application image is created and before the DMG or PKG installer is built. No default script is provided.

macOS DMG

- DMG setup script, *`application-name`*`-dmg-setup.scpt`.
- Applications license properties list, *`application-name`*`-license.plist`.
- Background file, *`application-name`*`-background.tiff`.
- Drive icon, *`application-name`*`-volume.icns`.

macOS PKG

- Pre-installation script, `preinstall`. Script that is run before the application is installed.
- Post-installation script, `postinstall`. Script that is run after installation completes.
- Background image for Light Mode, *`application-name`*`-background.png`.
- Background image for Dark Mode, *`application-name`*`-background-darkAqua.png`.

Windows

- Post-image script, *`application-name`*`-post-image.wsf`. Custom script that is executed after the application image is created and before the MSI installer is built for both `.msi` and `.exe` packages. No default script is provided.
- Main WiX source file, `main.wxs`.
- WiX source file with WiX variables overrides, `overrides.wxi`. Values in this file override values in the main WiX file.

- Icon file, *launcher*.ico, for the main launcher and any additional launchers. More than one file can be provided. The file name must match the name of the application or the name of a launcher. If an icon file is not provided for a launcher, the default icon is used.
- Launcher properties file, *launcher*.properties.

View Resources

You can use the --verbose and --temp options for jpackage to get information about the resources used to package your application.

To decide if you need to override the jpackage resources, review the current defaults:

- Use the --verbose option to see what is currently used.

  The --verbose option provides detailed information about the process of creating the package. The information also includes instructions for customizing the resource, such as the name of the file to add to the resource directory.

  The following example shows the jpackage command run on Windows to package the Dynamic Tree application, followed by snippets of the output from the --verbose option that show the default resources used. Note that to override the WinLauncher.template resource, a file named DynamicTree.properties is needed; to override the main.wxs resource, a file named main.wxs is needed

```
jpackage --input DynamicTree --main-jar DynamicTree.jar --verbose
WARNING: Using incubator modules: jdk.incubator.jpackage
Running [candle.exe, /?]
Running [C:\Program Files (x86)\WiX Toolset v3.11\bin\candle.exe, /?]
Windows Installer XML Toolset Compiler version 3.11.1.2318

   ...

Using default package resource java48.ico [icon] (add DynamicTree.ico to
the resource-dir to customize).
Using default package resource WinLauncher.template [Template for creating
executable properties file]
(add DynamicTree.properties to the resource-dir to customize).

   ...

Using default package resource main.wxs [Main WiX project file] (add
main.wxs to the resource-dir to
customize).
Using default package resource overrides.wxi [Overrides WiX project file]
(add overrides.wxi to the
resource-dir to customize).

   ...
```

- Use the --temp option to keep temporary files for review.

  The --temp option provides jpackage with the name of a new or empty directory where temporary files are written during the packaging process. The path passed to jpackage can be absolute or relative to the current directory. When this option is used, the directory is not deleted at the end of the process.

Review this directory to see the resources that were used to package your application. Review each file to identify the properties and values that you might want to override. If you override a file, your custom file must contain all of the properties that the default contains.

The following example shows the directory created on Windows. The `config` directory contains resources that you can override.

```
jpackage --input DynamicTree --main-jar DynamicTree.jar \
   --temp DTtempfiles

\DTtempfiles
   \config
      DynamicTree.ico
      DynamicTree.properties
      main.wxs
      MsiInstallerStrings_en.wxl
      MsiInstallerStrings_ja.wxl
      MsiInstallerStrings_zh.wxl
      overrides.wxi
   \images
   \wixobj
```