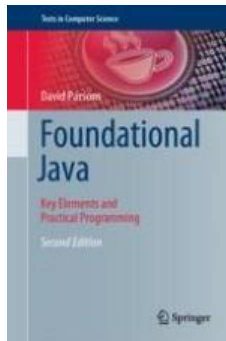


Chapter 14
Automatic Building and Testing with Ant

Foundational Java

Key Elements and Practical Programming

What is Ant?

- Ant (‘Another Neat Tool’) is a Java-based build tool which uses a combination of Java and XML to create platform independent build and deploy scripts.
- Being written in Java it works on any platform.
- It is open source and is available free from the Apache Software Foundation at http://ant.apache.org/
- It uses an XML build file (‘build.xml’ by default) containing *targets* and *tasks*

Ant

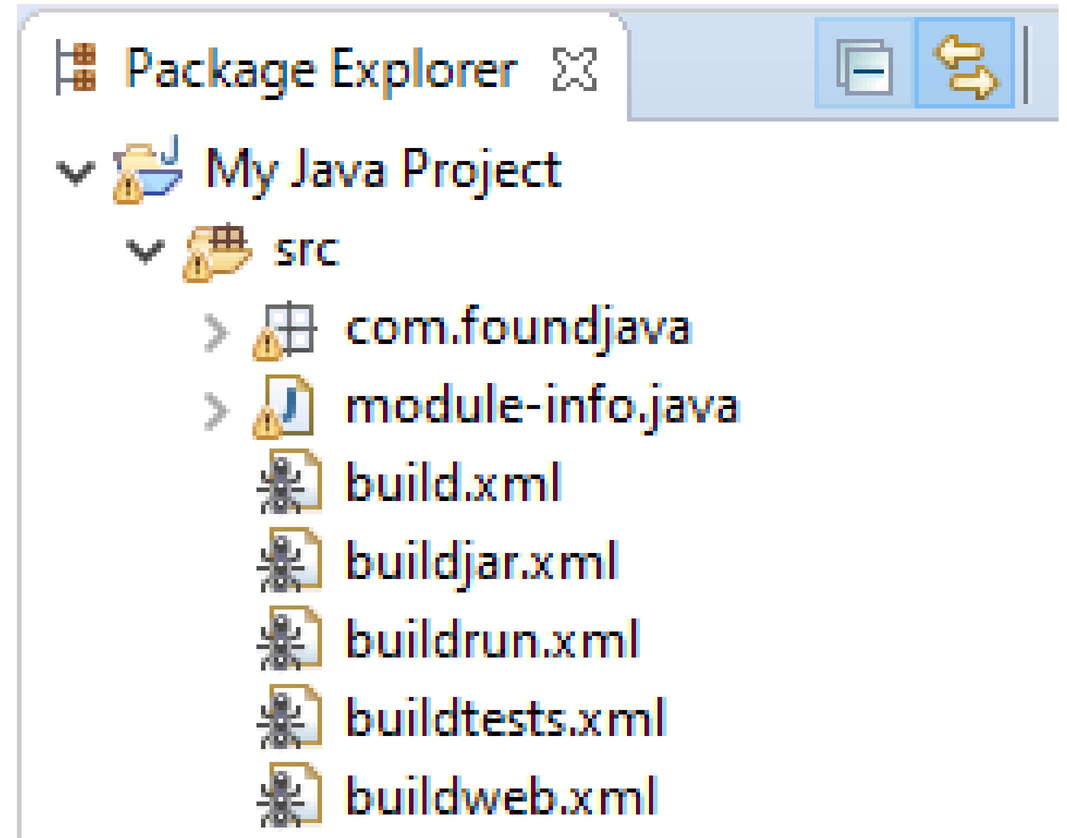
- Ant is used in many Java projects
- Ant can be invoked from many well-known tools, including Eclipse
- Has spawned subprojects such as Ivy (a dependency manager)
- Similar tools include Maven and Gradle, but Ant is ideal for relatively simple builds

Automating Development Processes

- Manually building and testing in Eclipse is not very efficient
- Also need to build and test outside of Eclipse
- Other tasks required
  - Packaging it into Java Archive (JAR files)
  - Deploying to an environment
- Ant provides a simple way of automating all of these processes

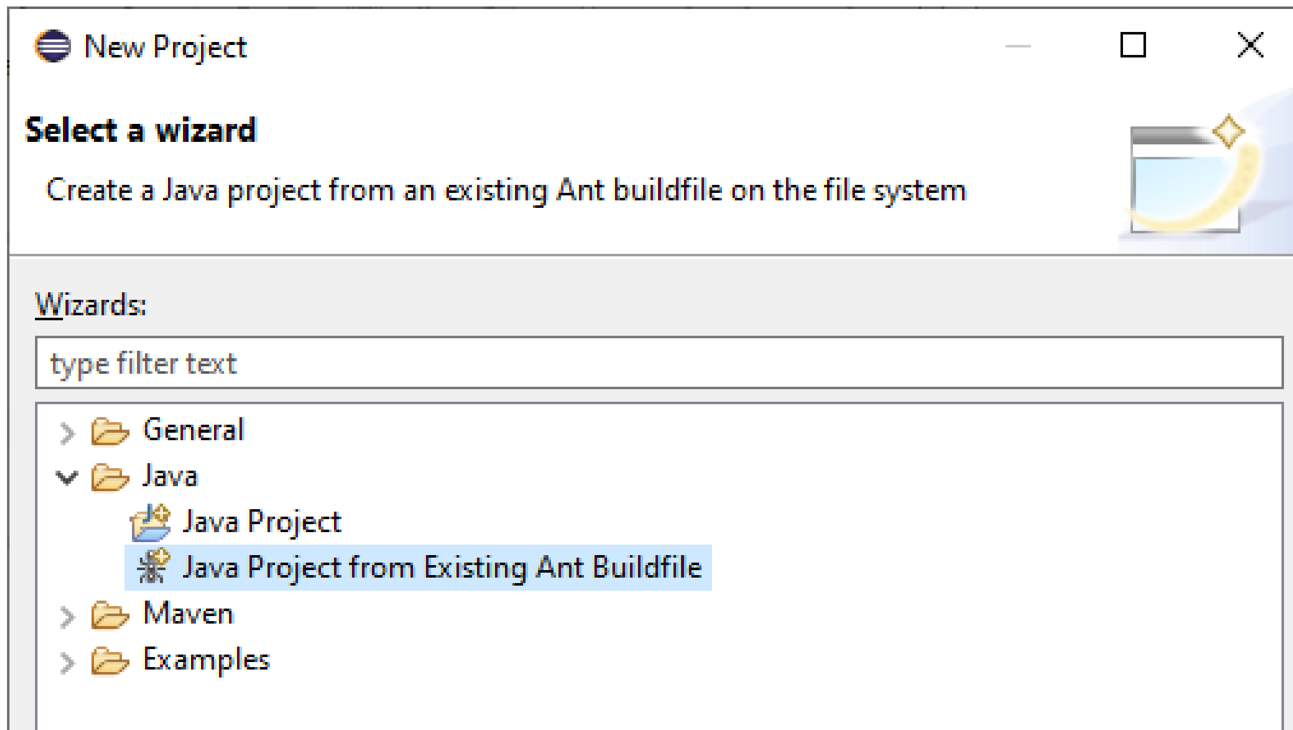
Using Ant in Eclipse

- No external configuration is needed
  - It is a standard component of the IDE
- To create a new Ant build script
  - Select 'New' -> 'File' from the pop-up or 'File' menu
  - Make sure the file is in the 'src' folder
  - Name the file 'build.xml'
  - Eclipse will treat this as an Ant file



Eclipse Project from Ant File

- Eclipse can also build a new project from an existing Ant ‘build.xml’ file (but assumes there is a javac task in it)
  - The optional project name is needed if building a new Eclipse project from the build file



The Ant Build File: ‘build.xml’

- An Ant build file can specify a series of tasks to be performed, such as
  - Removing and recreating output folders to ensure a clean build
  - Compiling Java source code
  - Creating Java Archive (JAR) files or other deployment formats
  - Copying files to distribution locations
  - Running Java applications
  - Running JUnit tests

Properties, Tasks and Targets

- ‘build.xml’ consists of a project
- Comprises a set of *tasks*
  - Compiling, copying etc.
- Put into named *targets* that can be invoked by names
- Can also specify *properties*
  - Aliases for named resources

The ‘project’ Element

- The root ‘project’ element has three attributes:
  - ‘name’ (of project)
  - ‘default’ (target when not specified - required)
  - ‘basedir’ (directory from which all path calculations are done - optional, if not stated uses build file directory)

```
<project name=“Ant Project" default="compile“ basedir=".">
  <!--
    build file properties and targets in here
    there must be a target with the default name
  -->
</project>
```

Ant Properties

- A project can have a set of properties
- A property has name and value attributes

```
<property name="build" value="C:\javasource"/>
```

- Properties are usually listed at the top of the build file (i.e. before they are needed)
- Properties may be used within targets by placing the name between “${“ and “}”:

```
${propertyname}
```

- e.g.

```
${build}
```

Tasks

- Built-in tasks can be used in every Ant build file, e.g.

Task	Action
mkdir	creates directories
delete	removes files and directories
copy	copies files and directories
javac	invokes the java compiler
java	invokes the java vm
jar	creates Jar files
junit	runs JUnit tests

Targets and Tasks

- A target contains one or more tasks
- A task is defined as an XML element in the build file
- Tasks may have attributes and/or additional nested elements
- The ‘mkdir’ task is a simple empty element

```
<mkdir dir="directory_name" />
```

- The ‘javac’ task, which compiles Java source code, has two attributes to specify the source and destination directories

```
<javac srcdir="source_dir" destdir="destination_dir"/>
```

Target Elements

- A target must have a ‘name’ attribute
- All targets can have an optional ‘description’ attribute
- This ‘prepare’ target contains the ‘mkdir’ task to create a directory (note the use of a previously defined ‘build’ property name)

```
<target name="prepare" description="create a build folder">
  <mkdir dir="${build}" />
</target>
```

- This target, called ‘compile’, uses the ‘javac’ task to compile Java files. Again, property names are being used.

```
<target name="compile" description="compile source code">
    <javac srcdir="${src}" destdir="${build}" includeantruntime="false"/>
</target>
```

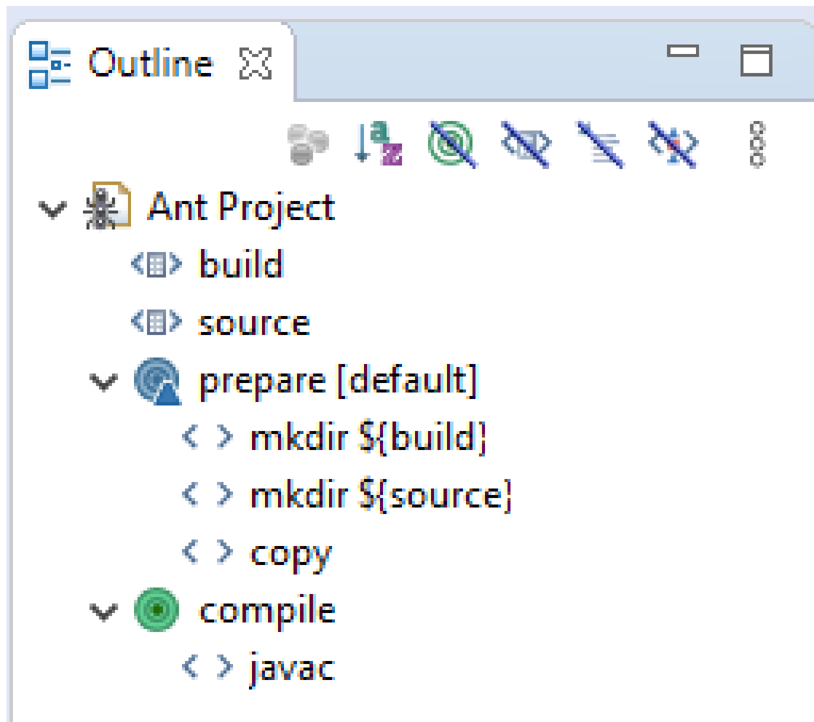
Example Build File

- Creates folders, copies a file and compiles it

```
<project name="Ant Project" default="prepare" basedir=".">
  <property name="build" value="c:\AntProject\javabuild"/>
  <property name="source" value="c:\AntProject\javasource" />

  <target name="prepare" description="create folders and copy source code">
    <mkdir dir="${build}"/>
    <mkdir dir="${source}"/>
    <copy todir="${source}"
      file="com\foundjava\chapter6\Die.java"/>
  </target>

  <target name="compile"
      description="compile all source code">
    <javac srcdir="${source}" destdir="${build}"  includeantruntime="false"/>
  </target>

</project>
```

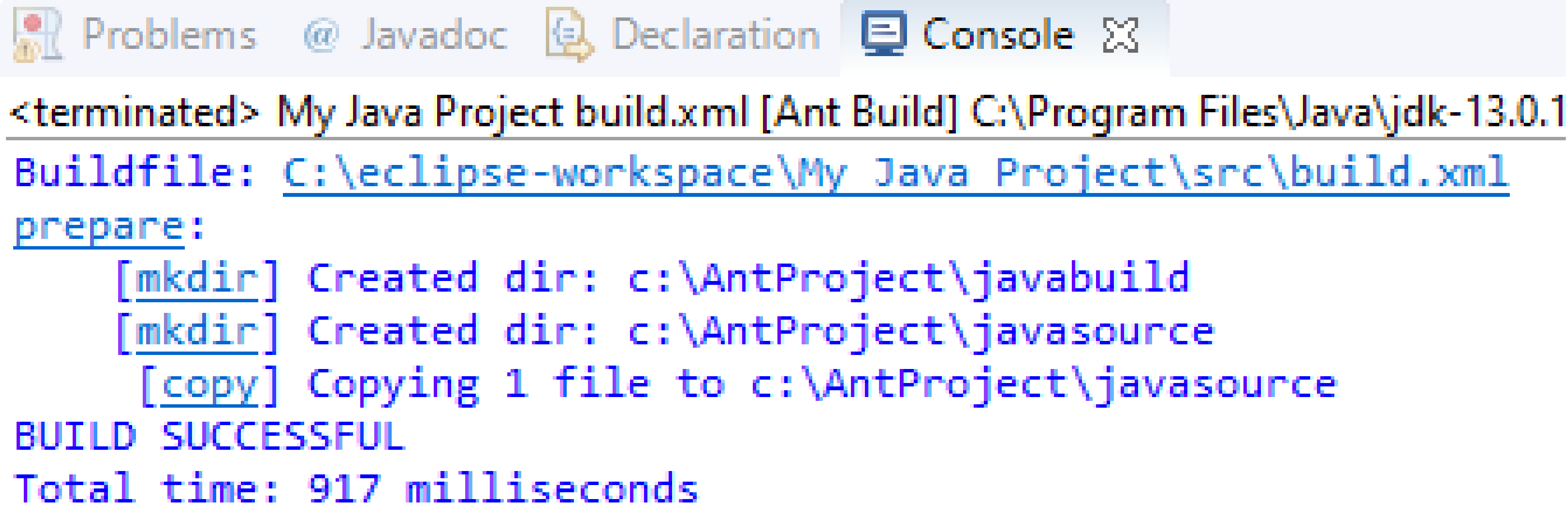
The Outline Pane

- The ‘outline’ pane will show the various properties, targets and tasks in a build file
  - Indicates the default target
  - The project and its targets can be expanded or collapsed to modify the view



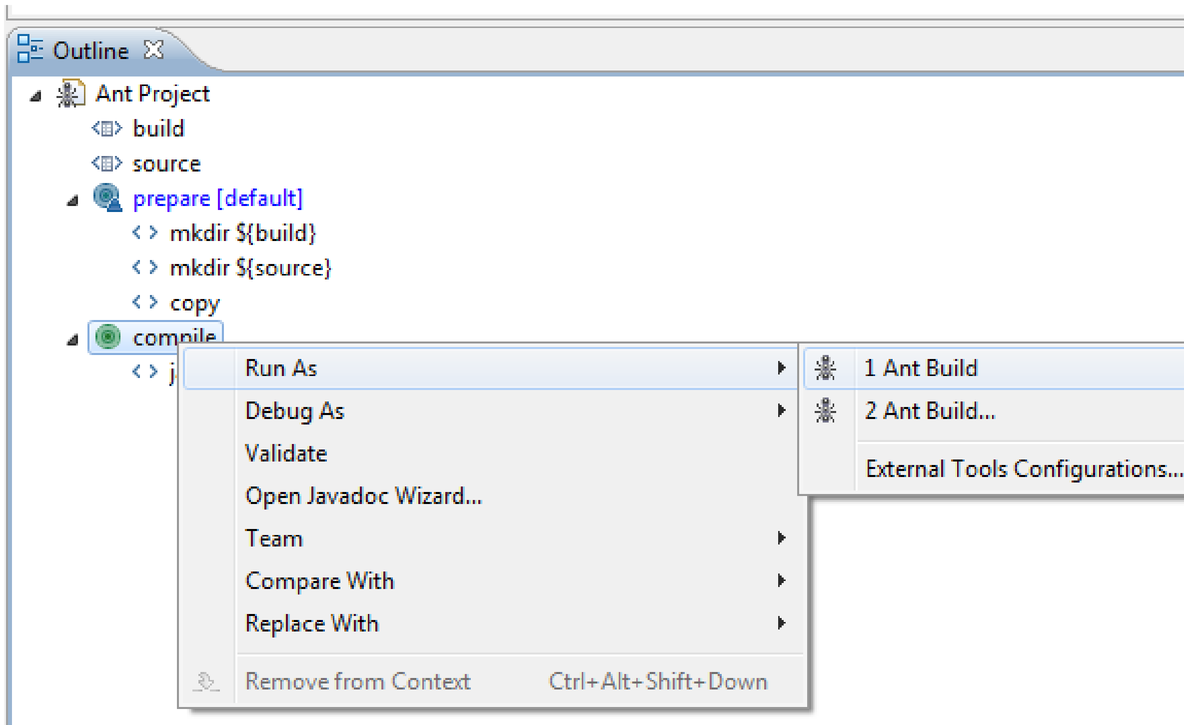
Running an Ant Build File in Eclipse

- Right click on the file
  - ‘Run As’ -> ‘Ant Build’
- The output from the build script will appear in the console



Running a Non-Default Target

- In the Outline window, any task can be run as an Ant Build
  - Right click on the target
  - Select 'Run As' -> 'Ant Build'



Exercise 14.1

- Create an Ant build file in Eclipse to copy the source code of the MyJavaProgram class from Chapter 2 to a folder outside of Eclipse, and compile it so that the byte code is written to another folder
- Run the Ant build and check that the compiled byte code is in the expected folder
- Run the compiled code from the command line

Copying Multiple Files

- Ant provides a nested “fileset” element inside the “copy” element to make it possible to copy all the files in a folder
- In this example, all the files in the “com.foundjava.chapter8” folder are copied to the source folder

```
<copy todir="${source}">
  <fileset dir="com\foundjava\chapter8"/>
</copy>
```

Packaging Code with the Ant ‘jar’ Task

- The Ant ‘jar’ task builds a JAR file
  - ‘jarfile’ attribute defines the name of the output file
  - ‘basedir’ defines the location of the code to be put into the JAR

```
<property name="dist" value="c:\AntProject\javadist" />

<target name="package">
   <jar jarfile="${dist}/mycode.jar" basedir="${build}" />
</target>
```

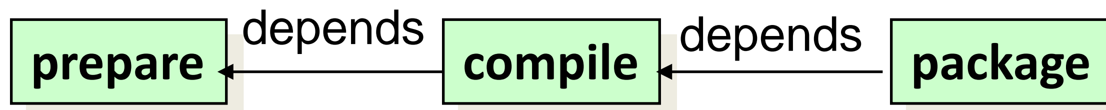
Runnable JAR Files

- A JAR file can be run directly, using a specific class as the program entry point
- A runnable JAR can be created by including the nested ‘manifest’ element to set the ‘Main-Class’ entry

```
<target name="package">
  <jar destfile="${dist}/mycode.jar" basedir="${build}">
    <manifest>
      <attribute name="Main-Class" value="com.introjava.chapter8.DrawFrame" />
    </manifest>
  </jar>
</target>
```

Target Dependencies

- Often one target will depend on another being executed first
- We can specify these dependencies with the ‘depends’ attribute of the ‘target’ element
- For example we might want to ensure that the code is compiled before the ‘jar’ file is created
  - the ‘package’ target depends on the ‘compile’ target
- And in turn that directories are prepared before the source code is compiled
  - the ‘compile’ target depends on the ‘prepare’ target



Dependency Example

- We can implement this chain of dependencies by making one target depend on another

```
<target name="compile" depends="prepare" description="compile all source code">
  <javac srcdir="${source}" destdir="${build}” includeantruntime="false"/>
</target>
```

- If a target depends on another target, then Ant will execute the other target first.
  - <target name="package" depends="compile">
  - <target name="compile" depends="prepare">

Complete Build File (1)

```
<project name="Ant Project" default="package" basedir=".">
  <property name="build" value="c:\AntProject\javabuild" />
  <property name="source" value="c:\AntProject\javasource" />
  <property name="dist" value="c:\AntProject\javadist" />

  <target name="prepare"
      description="create output folders for the build">
    <delete dir="${build}" />
    <delete dir="${source}" />
    <delete dir="${dist}" />
    <mkdir dir="${build}" />
    <mkdir dir="${source}" />
    <mkdir dir="${dist}" />
    <copy todir="${source}">
     <fileset dir="com\foundjava\chapter8"/>
    </copy>
  </target>
```

Complete Build File (2)

```
  <target name="compile" depends="prepare"
      description="compile all source code">
    <javac srcdir="${source}" destdir="${build}"
      includeantruntime="false"/>
  </target>

  <target name="package" depends="compile">
    <jar destfile="${dist}/mycode.jar" basedir="${build}">
      <manifest>
        <attribute name="Main-Class"
          value="com.foundjava.chapter8.DrawFrame" />
      </manifest>
    </jar>
  </target>

</project>
```

Target Dependencies in Action

- Calling Ant using the 'package' target will generate console output that shows every target and the tasks invoked

Problems @ Javadoc Declaration Console

```
<terminated> My Java Project buildjar.xml [Ant Build] C:\Program Files\Java\jdk-13.0.1
Buildfile: C:\eclipse-workspace\My Java Project\src\buildjar.xml
prepare:
   [delete] Deleting directory c:\AntProject\javabuild
   [delete] Deleting directory c:\AntProject\javasource
    [mkdir] Created dir: c:\AntProject\javabuild
    [mkdir] Created dir: c:\AntProject\javasource
    [mkdir] Created dir: c:\AntProject\javadist
     [copy] Copying 24 files to c:\AntProject\javasource
compile:
    [javac] Compiling 24 source files to c:\AntProject\javabuild
package:
      [jar] Building jar: c:\AntProject\javadist\mycode.jar
BUILD SUCCESSFUL
Total time: 2 seconds
```

Output shows every target and the tasks invoked

Running the 'package' Ant Target

The Manifest, and Running a JAR

- The MANIFEST.MF file in the generated JAR will include the 'Main-Class' entry

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.10.7
Created-By: 13.0.1+9 (Oracle Corporation)
Main-Class: com.foundjava.chapter8.DrawFrame
```

- A JAR file with a main class can be run directly from the command line using the 'jar' option on the Java run time

```
C:\AntProject\javadist>java -jar mycode.jar
```

  - Ensure that the PATH has been set to the "bin" folder of your Java installation prior to running the "java" command.

Running Code, Forking and Classpaths

- Ant can also be used to run code
- The ‘java’ task can be used to run any class with a ‘main’ method, but requires some configuration
  - classpath and pathelement
  - ‘fork’ allows the Java run time to be launched in a separate VM

```
<target name="run" depends="compile">
  <java classname="com.introjava.chapter8.DrawFrame" fork="true">
    <classpath>
      <pathelement location="${build}" />
    </classpath>
  </java>
</target>
```

Running a JAR With a ‘java’ Task

- The ‘java’ task can also be used to run a JAR file, as long as that JAR file has a main class defined in its manifest

```
<target name="runjar" depends="package">
  <java jar="${dist}/mycode.jar" fork="true" />
</target>
```

Exercise 14.2

- Modify your Ant build file from Exercise 14.1 by adding a target to run the class
- Add another target to package the compiled class into a JAR file
- Add a further target to run the JAR file
- Add the required dependencies between targets
- Run the build file and check that all the steps complete successfully.
- Run your JAR file from the command line

Running Tests

- As well as building and running code, we can use Ant to run JUnit tests
- Ant has a special “junitlauncher” task that can be used to run selected test classes
- In this example, source code comes from the “com.foundjava.chapter10” package,
  - There are dependencies on classes from other packages so these are included in the source files/filesets to ensure successful compilation.
  - The test suite file is deleted here due to complex dependencies

```
<copy todir="${source}"  file="com\foundjava\chapter6\Die.java"/>
  <copy todir="${source}">
    <fileset dir="com\foundjava\chapter8" />
    <fileset dir="com\foundjava\chapter9" />
    <fileset dir="com\foundjava\chapter10" />
  </copy>
  <delete file="${source}\ShapeSuite.java"/>
</target>
```

Setting the Classpath with a ‘path’ Element

- It can be helpful to specify a reusable classpath using the ‘path’ task
- The ‘path’ task may contain multiple ‘pathelements’, each one referring to a different path or file
- This example includes both the build directory and the location of all the jar files that Junit requires

```
<path id="project.classpath">
  <pathelement path="c:\junit5\junit-jupiter-api-5.5.2.jar" />
  <pathelement path="c:\junit5\junit-platform-launcher-1.6.0.jar" />
  <pathelement path="c:\junit5\junit-platform-commons-1.6.0.jar" />
  <pathelement path="c:\junit5\junit-platform-engine-1.6.0.jar" />
  <pathelement path="c:\junit5\junit-jupiter-engine-5.5.2.jar" />
  <pathelement path="c:\junit5\apiguardian-api-1.0.0.jar" />
  <pathelement path="c:\junit5\opentest4j-1.2.0.jar" />
  <pathelement path="${build}" />
</path>
```

Using a Classpath

- The ‘id’ of the path provides a unique identifier within the build file that can be used by other tasks
- Here, a ‘javac’ task includes a reference to the classpath that has previously been defined, using the nested ‘classpath’ element
  - If ‘javac’ includes a ‘classpath’ it is no longer an empty element, but has the ‘classpath’ element nested inside it
  - The ‘refid’ attribute refers to the id of a previously defined ‘path’

```
<target name="compile" depends="prepare" description="compile all source code">
  <javac srcdir="${source}" destdir="${build}"  includeantruntime="false">
    <classpath refid="project.classpath" />
  </javac>
</target>
```

Running Tests with the “junitlaucher” and “test” Elements

- The “junitlauncher” task has many attributes
  - Most have default values.
  - “test” elements can be used to run individual test classes
  - The only required attribute for the “test” element is “name” (the name of the JUnit test class).
  - “project.classpath” is included in the “classpath” element.
  - “printsummary” shows a summary of the tests that have been run
  - “haltonfailure” will halt the tests if a failure occurs.

```
<target name="junit" depends="compile" description="run JUnit 5 tests">
  <junitlauncher printsummary="true" haltonfailure="true">
    <classpath refid="project.classpath"/>
    <test name="com.foundjava.chapter10.RectangleTestCase"/>
    <test name="com.foundjava.chapter10.CircleTestCase"/>
  </junitlauncher>
</target>
```

Running a JUnit Test

- A JUnit test being run by an Ant script (in the 'junit' target)

```
Console
<terminated> My Java Project buildtests.xml [Ant Build] C:\Program Files\Java\jdk-13.0.1\bin\javaw.exe
Buildfile: C:\eclipse-workspace\My Java Project\src\buildtests.xml
prepare:
   [delete] Deleting directory c:\AntProject\javabuild
   [delete] Deleting directory c:\AntProject\javasource
   [delete] Deleting directory c:\AntProject\javadist
    [mkdir] Created dir: c:\AntProject\javabuild
    [mkdir] Created dir: c:\AntProject\javasource
    [mkdir] Created dir: c:\AntProject\javadist
     [copy] Copying 1 file to c:\AntProject\javasource
     [copy] Copying 49 files to c:\AntProject\javasource
   [delete] Deleting: c:\AntProject\javasource\ShapeSuite.java
compile:
    [javac] Compiling 49 source files to c:\AntProject\javabuild
junit:
[junitlauncher] Test run finished after 194 ms
[junitlauncher] [         2 containers found      ]
[junitlauncher] [         0 containers skipped    ]
[junitlauncher] [         2 containers started    ]
[junitlauncher] [         0 containers aborted    ]
[junitlauncher] [         2 containers successful ]
[junitlauncher] [         0 containers failed     ]
[junitlauncher] [         1 tests found           ]
[junitlauncher] [         0 tests skipped         ]
[junitlauncher] [         1 tests started         ]
[junitlauncher] [         0 tests aborted         ]
[junitlauncher] [         1 tests successful      ]
[junitlauncher] [         0 tests failed          ]
[junitlauncher] Test run finished after 105 ms
[junitlauncher] [         2 containers found      ]
[junitlauncher] [         0 containers skipped    ]
[junitlauncher] [         2 containers started    ]
[junitlauncher] [         0 containers aborted    ]
[junitlauncher] [         2 containers successful ]
[junitlauncher] [         0 containers failed     ]
[junitlauncher] [         1 tests found           ]
[junitlauncher] [         0 tests skipped         ]
[junitlauncher] [         1 tests started         ]
[junitlauncher] [         0 tests aborted         ]
[junitlauncher] [         1 tests successful      ]
[junitlauncher] [         0 tests failed          ]
BUILD SUCCESSFUL
Total time: 3 seconds
```

Exercise 14.3

- Design the tasks and targets for an Ant build file to build the Die class and the DieTestCase class from Chap. 10.
- Include a target that will run your JUnit test.
- Add one of your own test cases from the exercises into the build.

Installing Ant (standalone)

- Ant is downloaded as a zip file that simply needs to be extracted
- Set up two environment variables
  - JAVA_HOME, to reference your Java installation
  - ANT_HOME, to reference your Ant installation
- Add ANT_HOME\bin to the system path
- You can invoke Ant via ant.bat
  - By default, looks for a ‘build.xml’ file in the local folder
- If you want to run JUnit tests with an external installation of Ant, you can add all the required JAR files to the “lib” folder of your Ant installation.

Summary

- Ant build tool
  - Automate the build and deploy process
  - Run and test code
- Build file can take source code out of Eclipse and build it independently of the IDE
  - Still able to run build scripts from within Eclipse
- Configuration can be applied to running JUnit tests from Ant
  - Including options to control what is written to the console