SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

Topic Notes: Java Packages

As Java programs become more complicated, they bring together code and data broken down into more and more classes. Some of these are written specifically for a given program, but many are reusable classes that are likely to be brought in from libraries, either the standard Java API classes, or classes from other sources.

Most modern programming languages have mechanisms that help to manage this. Java provides the *package* system. A Java package allows collections of related Java classes to be grouped. Since Java 9, packages in the Java API are further grouped into *modules* that contain related Java packages. We will focus on how classes are organized into packages.

Whether you have thought about it or not, you've been using classes from packages since some of your first Java programs. Classes like `Integer` and `System` are part of the package called `java.lang`. The classes in this package are available to all Java programs.

A list of all of the classes (and other entities like interfaces and enuerated types) in the `java.lang` package can be found by going to the Java API documentation for one of the classes in that package, and clicking on the link near the top of the page to Package `java.lang`.

Despite the fact you've used this classes in nearly every Java program you've ever written, there's a good chance you never have typed "`java.lang`" into any Java program. That's because the classes in `java.lang` are always imported to Java programs. Classes from other packages require explicit `import` statements.

Other Java API Packages

You have also used classes from other Java API packages regularly in your programs.

The `java.util` package includes commonly-used classes like `ArrayList`, `Scanner`, and `Random`.

You've regularly used classes from other Java packages as well, such as `java.io`, `java.text`, `java.awt`, and more.

When using these, as you know, you need to add `import` statements to the top of your program.

You can import all classes in a given package:

```
import java.util.*;
```

or only those you intend to use:

```
import java.util.ArrayList;
import java.util.Scanner;
```

Early in the semester, we discussed the advantages of the latter specification, and we will revisit that soon.

Any computer capable of compiling and running Java programs should be able to import classes from the Java standard API without additional work.

Other packages can come from external sources, and might require additional installation of the package or packages (often as a "Jar File", which we will look at later) and configuration of your Java IDE and/or run-time system.

As an example, consider the Java packages in the Apache Commons: `https://commons.apache.org/`

Notice that the packages here have names that start with the Internet domain name of the site, but in reverse order. So the `SimpleEmail` class is specified by its *fully qualified name* `org.apache.commons.mail.SimpleEmail`.

If we had a program that wished to use this class, we could import the entire package:

```
import org.apache.commons.mail.*;
```

or import just that one class that will be used:

```
import org.apache.commons.mail.SimpleEmail;
```

or, the program could use the fully qualified name every time the class name is needed in the program:

```
org.apache.commons.mail.SimpleEmail e =
   new org.apache.commons.mail.SimpleEmail();
```

and not have any `import` statement for it at all.

Not all packages follow the domain name convention. The structure package, which some of you might have encountered in previous courses, and which we looked at briefly in this class as an example of a good class hierarchy, simply places all of its classes and interfaces into the `structure5` package. Convention would suggest that the package's fully qualified name should be `edu.williams.cs.structure5`. However, as it is intended as an educational package rather than something to be used in production, commercial projects, this is not likely to be problematic.

Using Packages

Java packages allow developers to group related classes together.

Java packages allow multiple classes with the same name to be used together in a single Java program. For example, within the Java API, the name `Timer` refers to (at least) three different classes: `java.util.Timer`, `javax.management.timer.Timer`, and `javax.swing.Timer`. `Timer` is a perfectly reasonable name for each of these classes. But consider a program that has need to use a `java.util.Timer`, but also uses some classes from `javax.swing`.

`https://github.com/SienaCSISAdvancedProgramming/TwoTimers`

Which class does the name `Timer` refer to in `main`?

Try to compile it to find out how Java handles this.

To avoid ambiguity, we could explicity construct a `java.util.Timer` object. But better yet, we should avoid this conflict in the first place here by not using *wildcard imports*.

Including lines like

```
import java.util.*;
```

in your programs as a beginning programmer is generally accepted, and allows beginners not to worry too much about the `import`s needed for their `Scanner`s and `Random`s and other classes they are using.

However, as programs become more complex, it is the best practice to avoid wildcard imports and to list every class or interface you need on a separate `import` statement line.

In those cases where ambiguity remains: suppose your program really did need to use two different `Timer` objects, one a `java.util.Timer` and one a `javax.swing.Timer`, you would need to use their fully qualified names (which would allow you to omit the `import` statements, if you wished).

Protection Levels

Now that we are thinking about Java code as organized into packages, we can more fully understand the protection levels Java provides for classes, methods, and variables.

Read `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html` for the official explanation.

Creating your own Packages

Most of the code you have written to this point is likely in the *default package*. This is fine in many cases, but at times you will want to write code that you put into your own packages.

This is accomplished using the `package` keyword. Many of you have seen this keyword introduced into your programs by BlueJ when you have `package.bluej` files in multiple places in a directory hierarchy.

When developing large projects, breaking the Java code into packages that contain related classes and interfaces is helpful for code organization.

A very important time to use a package, however, is when developing a set of classes and interfaces that are intended to be generally useful to many other Java programs.

As an example, we will look at a small package that provides a couple of classes that will "factor out" some of the boilerplate code we've seen over and over in our programs that use graphical animations, and a version of one of our earlier examples that makes use of these classes.

`https://github.com/SienaCSISAdvancedProgramming/ThreadGraphicsPackage`

The first thing to notice in this repository is the directory structure necessitated by the fact that we are using a properly qualified package name for the abstract classes. The package name is `edu.siena.csis225.threadgraphics`, so inside of a `src` directory in the repository, there is a path to the code `edu/siena/csis225/threadgraphics`.

In that directory, we find the two classes that make up this package: `ThreadGraphicsController` and `AnimatedGraphicsObject`.

The first line in each

```
package edu.siena.csis225.threadgraphics;
```

indicates that this class is part of the given package.

`ThreadGraphicsController` is a concrete base class that implements the common functionality we often see in the class that has a `main` method to start the program and a `run` method to set up the GUI, including a `JPanel` with an overridden `paintComponent` method that draws our graphics. It also includes code to manage a list of `AnimatedGraphicsObject` objects, drawing them when necessary and removing them from the list when they're done with their animations.

`AnimatedGraphicsObject` is an abstract class that represents the individual animated graphics objects.

There are some explanatory comments in each class. Here are some particular things to notice in each, some of which are specific to these being in a package, others general comments about the design of these classes.

For `ThreadGraphicsController`:

- We can implement `Runnable` as usual, but do not extend a class like `MouseAdapter`, since as a more general purpose base class, we don't know which mouse methods, if any, it will use. Or maybe it uses keyboard events. So those details will be left to the derived class.
- The first two instance variables are ones we have seen in the corresponding class of many of our graphical animation examples. We have a list of animated objects, in this case of type `AnimatedGraphicsObject` that we will look at next, and the `JPanel` in which our graphics will be drawn.

- We'll come back to the next two instance variables when we look at the constructor.
- We saw recently that our list of animated graphical objects might be modified not only in our `paintComponent` method, but also by event handlers, and that this can lead to race conditions. So we introduce an instance variable `lock` to serve as an explicit lock protecting the critical sections where the `list` variable might be modified.
- We'll also come back to the `thisTGC` variable.
- The constructor is needed here to accept the window label and size that we've been hard-coding into the `run` method of each of our applications. Instead, we will have the derived class that knows what the window should be named and how big we would like it to be, call this constructor to pass that information in. It's stored in instance variables, and used just below near the start of the `run` method.
- Much of the rest of the `run` method looks very familiar from all of our copying and pasting from example to example (a leading indicator that inhertiance is needed), but there are a few items that depend on the specifics of the application that we need to deal with.
  - In the `paintComponent` method, we sometimes need to draw some static background (*e.g.*, the scene in the falling snow) or need to have some graphical feedback for mouse events (*e.g.*, the sling line in the ball tosser or the rubber-banding lines and triangles in the Sierpinski gasket). These depend on the actual application, so these are to be done by a `paint` method, defined below. Note: since the `paintComponent` method is a method of the `JPanel`, not the `ThreadGraphicsController`, the `this` reference in scope here refers to the `JPanel`. We work around this by storing the `tgc`'s `this` reference in an instance variable `thisTGC` for use here.
  - The rest of `paintComponent` is our now-familiar loop to visit each animated object and either paint it, or remove it from the list if it's done.
  - Back in the `run` method, we have some tasks that again are application-dependent. So we move them into methods that can be overridden by the derived class, which knows the details. `buildGUI`, by default, adds the panel to the frame. But we know that some applications have Swing components and hierarchies of panels. In those cases, the application would override `buildGUI`. Similarly, different applications use different event handlers, and those event handlers need to be added. This is done by overriding the default (empty) `addListeners` method.
  - The `run` method wraps up with the construction of our list, and the last couple lines to get the window displayed.
- The remainder of `ThreadGraphicsController` has the default implementations of the three methods mentioned above: `paint` and `addListeners`, which do nothing by default, and `buildGUI`, which just adds the panel to the frame by default.

For `AnimatedGraphicsObject`:

- Since there is so much variety in what animated objects look like and what "animation" means in different contexts, there is not as much we can factor out in `AnimatedGraphicsObject`.
- We extend `Thread` here, since all of our animated objects would do so, and they'll need to extend `AnimatedGraphicsObject`.
- We have seen two instance variables in all of our animated objects, `done` and `container`, and those are declared here. Note that we change their protection level to `protected` so they can be accessed by the derived classes.
- The `AnimatedGraphicsObject` constructor just sets the `container`. Derived classes should either call this constructor from theirs or have their constructor set `container`.
- The next method is a convenience method that we can call to have a thread sleep without having to wrap it up in a `try-catch` block. Note that it is declared as `static`, since it does not need access to instance variables.
- We include the `done` method we have seen in pretty much every animated object.
- We know all `AnimatedGraphicsObject` objects will need a `paint` method, as it's called from `ThreadGraphicsController`'s `run` method on each object in the list. But we know nothing about what the `paint` method needs to do, so it's declared as `abstract`.
- Finally, we override the `Thread`'s `run` method with an `abstract` run method here, forcing derived classes to provide one.

To compile the classes in our package, we will use the Java compiler at the command line. Normally, to do this, we would make sure our working directory is the same one that contains our source files, and issue the `javac` command. However, when working with packages, we need to have our working directory be at the top of the package's path. In this case, that's the `src` directory. Once there, we can compile the two Java files:

```
javac edu/siena/csis225/threadgraphics/*.java
```

We will see soon how we can package the resulting class files into a "Jar file", so we can use the package without having to put its source code tree into our project.

To see how these classes can help simplify a specific application, let's look in the `src/examples/BallTosser` directory. The program here has identical functionality to our earlier `BallTosser` example, but has been modified to use the code from our new package.

For convenience in this particular repository, the two files for the Ball Tosser program are in package `example.BallTosser`. Both then import all (2) classes from the package we just looked at:

```
import edu.siena.csis225.threadgraphics.*;
```

Now let's look at what has changed in the code between the original versions of these files and the ones that use the package.

First, `BallTosser`:

- The class now extends `ThreadGraphicsController` and implements the two mouse event interfaces. We can no longer extend `MouseAdapter`, since we need to extend `ThreadGraphicsController`. This only means we will need to provide dummy implementations of the mouse event handlers we aren't using. It no longer needs to implement `Runnable` because `ThreadGraphicsController` does that (though it would cause no harm to repeat it here).
- We are able to remove the `list` and `panel` instance variables, as those are inherited from `ThreadGraphicsController`.
- We introduce a constructor, which calls the `ThreadGraphicsController` constructor with appropriate parameters for this application.
- The `run` method is no longer needed, as the `ThreadGraphicsController` one we inherited has all of the needed functionality. However, we do need to provide implementations for two of the three methods `ThreadGraphicsController`'s `run` method calls.
  - `paint` is needed to draw our sling line while the mouse is being dragged.
  - `addListeners` adds the listeners for our mouse events.
  - We do not need to override `buildGUI`, as the default implementation is sufficient here.
- The mouse event handlers are unchanged, except that we need to provide the empty methods for the ones we don't use, to satisfy the interfaces.
- No change is needed to `main`.

And now, `BouncingGravityBall`:

- `BouncingGravityBall` now extends `AnimatedGraphicsObject` instead of `Thread`.
- We still need all of our constants and all except the two instance variables that are now defined in `AnimatedGraphicsObject`.
- The only change to the constructor is that instead of assigning `container` directly, we call the superconstructor to do it.
- `paint` is unchanged.
- The only change in `run` is to take advantage of the `sleepWithCatch` method.
- We do not need an implementation of `done`, it is inherited.

To compile this example, we again need to be in the `src` directory, and we can issue the command:

```
javac example/BallTosser/*.java
```

And finally we can run it from the `src` directory:

```
java example/BallTosser/BallTosser
```

Creating a Jar File

Java developers can share code they've developed, typically as packages, by creating and using *Jar Files* (**J**ava **AR**chive Files). These files group together a collection of `.class` files and can then be provided to the Java compiler and run-time system as a place to look to find other classes used by the program being compiled or executed.

To create a Jar file for our package, we can run the `jar` command from the command-line while in the `src` directory:

```
jar cf threadgraphics.jar edu/siena/csis225/threadgraphics/*.class
```

This command **creates** (the **c**) in a **file** (the **f**) named `threadgraphics.jar` an archive of all of the `.class` files specified in the remainder of the command. In this case, it's the three `.class` files generated from our two `.java` files that make up the package.

We can see what's archived in a file, also using the `jar` command, but with different parameters:

```
jar tf threadgraphics.jar
```

The output would look something like this:

```
META-INF/
META-INF/MANIFEST.MF
edu/siena/csis225/threadgraphics/AnimatedGraphicsObject.class
edu/siena/csis225/threadgraphics/ThreadGraphicsController$1.class
edu/siena/csis225/threadgraphics/ThreadGraphicsController.class
```

The first two lines indicate some housekeeping information that the Jar file uses to track its contents, then the last three list the three `.class` files that have been archived in this Jar file.

This Jar file can then be copied to other projects or added to the standard set of Jar files your development environment searches when you import and use classes outside of your project.

As an example of how this is done at the command line, we look at a new program that (among other things) demonstrates how to use the Jar file of the `threadgraphics` package in another project without copying over the source code.

`https://github.com/SienaCSISAdvancedProgramming/Breakout`

We will look at what this program is all about later, but for now, we focus on how it uses the Jar file, and how we need to change the way we compile things.

That repository has a copy of the `threadgraphics.jar` file. It also has an exciting implementation of a "Breakout" game, which has its `main` method in the class `Breakout`.

If we try to compile this project in our usual way (or if you tried to compile it in BlueJ), it will not work. On my Mac, the command:

```
javac *.java
```

gives 22 errors, one of the first of which is:

```
Breakout.java:1: error: package edu.siena.csis225.threadgraphics does not exist
import edu.siena.csis225.threadgraphics.*;
```

This makes sense. We haven't told `javac` that it should look in `threadgraphics.jar` for classes in addition to its usual places. Those "usual places" include the Jar files for the standard Java API, and our current directory.

We can add to the places to look by adding an option to our compile command:

```
javac -cp .:./threadgraphics.jar *.java
```

The `-cp` option says to replace the *classpath* with one that includes `.`, the Unix way to refer to the current directory, and the `threadgraphics.jar` file. Now, no errors!

The same situation arises when we try to run in our usual way with:

```
java Breakout
```

This gives an error due to a run-time exception:

```
Error: Could not find or load main class Breakout
Caused by: java.lang.NoClassDefFoundError: edu/siena/csis225/threadgraphics/ThreadC
```

Just as we needed to use a command-line parameter to `javac` so it could find the Jar file, we need to do this when we launch the Java Virtual Machine with the `java` command.

```
java -cp .:./threadgraphics.jar Breakout
```

If we want to avoid having to do that on each `java` and `javac` command, we can set an *environment variable* that will remember the new classpath.

```
export CLASSPATH=.:threadgraphics.jar
```

The above should work in Git Bash and in standard configuration Mac Terminal windows. The `java` and `javac` commands will check the value of this environment variable and use its value as the classpath if it exists.

Java IDEs typically have a way to specify that individual Jar files or entire directories of Jar files be included in their search path for a project. For example, in BlueJ, under the Preferences, you can go to the Libraries tab and add Jar files.