SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2017

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.

Whether you have thought about it or not, you've been using classes 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 at:

Java API Package Documentation: java.lang at
`http://docs.oracle.com/javase/8/docs/api/`

`/package-summary.html`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. Classes from other packages require more.

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`.

Java API Package Documentation: java.util at
`http://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html`

You've likely used classes regularly from other Java packages as well:

Java API Package Documentation: java.io at
`http://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html`

Java API Package Documentation: java.text at
`http://docs.oracle.com/javase/8/docs/api/java/tex/package-summary.html`

Java API Package Documentation: java.awt at
`http://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html`

Java API Package Documentation: javax.swing at
`http://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html`

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;
```

We will soon discuss the advantages of the latter specification.

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 study later in the semester) 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.mai`

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, 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 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`.

```
import java.util.*;import javax.swing.*;

public class TwoTimers {

    public static void main(String[] args) {

        new Timer();
    }

    // pretend there's some other stuff here that uses
    // things from javax.swing...
}
```

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

Try it out in:

See Example: TwoTimers

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. Get into this habit now.

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).

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. We will experiment with this during class by developing a small example in BlueJ.

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.

Obscuring

A final topic for now that is related to packages is the problem of *obscuring*. Consider this example:

See Example: Obscuring

We will talk in class about exactly what is going on here, but notice that `System.out.println` in `main` did *not* result in a call to the standard `System.out.println`!

A second example shows obscuring of the constant `MIN_PRIORITY` from the `Thread` class:

See Example: Obscuring2

Note that in both of these cases, we can still access the obscured name by using its fully qualified form.