CSCI 2120:
Software Design & Development II

Packages, Scopes, Naming Conventions

1. Introduction

- In this lecture, we'll cover the basics of packages in Java. We'll see how to create packages and access the types we place inside them.

- We'll also discuss naming conventions and how that relates to the underlying directory structure.

- Finally, we'll compile and run our packaged Java classes.

1. Introduction

In java, programmers can create several classes & interfaces. After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship. Thus, the classes and interfaces which handle similar or same tasks are put into the same directory or folder, which is also known as a package.

Packages act as "containers" for classes. A package represents a directory that contains related group of classes & interfaces.

2. Overview of Java Packages

In Java, we use packages to group related classes, interfaces, and sub-packages.

The main benefits of doing this are:

- Making related types easier to find – packages usually contain types that are logically related
- Avoiding naming conflicts – a package will help us to uniquely identify a class; for example, we could have a `edu.uno.csci2120.Application`, as well as `com.github.ted.Application` classes
- Controlling access – we can control visibility and access to types by combining packages and access modifiers
- Packages can be considered as data encapsulation (or data-hiding).

2. Overview - Why use Packages?

- Modularity & Encapsulation
- Packages are to Classes what Classes are to Methods
  - i.e. packages ← classes ← methods ← code

- Your class files are always contained within a package.
- If you do NOT explicitly name your package then your class files are implicitly in the unnamed package.

2. Overview - Why use Packages?

- Use packages to write clean and organized code

- Neatly organizes your subsystems of your application to maximize reusability, flexibility, and robustness.

2. Overview - Types of Packages

There are basically 2 types of java packages.

1. Predefined Packages (i.e. System Packages of Java API)
2. User-defined Packages

3. System Packages of Java API

As there are built in methods, java also provides inbuilt packages which contain lots of classes & interfaces. These classes inside the packages are already defined & we can use them by importing relevant package in our program.

Java has an extensive library of packages, a programmer need not think about logic or doing any task. For everything, there are the methods available in java that may be used by the programmer without developing the logic on their own.

3. System Packages of Java API

java.lang
Language Support classes. These are classes that java compiler uses & therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads, & exceptions.

java.util
Language Utility classes such as vector, hash tables, random, numbers, date, etc.

java.io
Input/Output support classes. They provide facilities for the input & output of data.

3. System Packages of Java API

java.awt
Set of classes for implementing graphical user interface. They include classes for windows, buttons, list, menus, & so on.

java.net
Classes for networking. They include classes for communicating with local computers as well as with internet servers.

3. System Packages of Java API

The JDK and other Java libraries also come with their own packages. **We can import pre-existing classes that we want to use in our project in the same manner.**

For example, let's import the Java core *List* interface and *ArrayList* class:

```
import java.util.ArrayList;import java.util.List;
```

We can then use these types in our application by simply using their name:

```
public class TodoList {
    private List<TodoItem> todoItems;

    public void addTodoItem(TodoItem todoItem) {
        if (todoItems == null) {
            todoItems = new ArrayList<TodoItem>();
        }
        todoItems.add(todoItem);
    }
}
```

Here, we've used our new classes along with Java core classes, to create a *List* of *ToDoItems.*

4. User Defined Packages

The users of the java language can also create their own packages. They are called user-defined packages. User defined packages can also be imported into other classes & used exactly in the same way as the built-in packages.

4. User Defined Packages

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

We can reuse existing classes from the packages as many time as we need it in our program.

4. User Defined Packages - How packages work

Package names and directory structure are closely related. For example if a package name is edu.uno.cs, then there are three directories, edu, uno and cs such that cs is present in uno and uno is present edu. Also, the directory cs is accessible through CLASSPATH variable, i.e., path of parent directory of cs is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

4. Creating a Package

To create a package, we have to use the package statement by adding it as the very first line of code in a file.

Let's place a type in a package named edu.uno.csci2120.packages:

```
package edu.uno.csci2120.packages;

public class className{

 //Body of className

}
```

4. Creating a Package

It's highly recommended to place each new type in a package. If we define types and don't place them in a package, they will go in the *default* or unnamed package. Using default packages comes with a few disadvantages:

- We lose the benefits of having a package structure and we can't have sub-packages
- We can't import the types in the default package from other packages
- The ***protected*** **and** ***package-private*** access scopes would be meaningless

As the Java language specification states, unnamed packages are provided by the Java SE Platform principally for convenience when developing small or temporary applications or when just beginning development.

4. Creating a Package - Naming Conventions

In order to avoid packages with the same name, we follow some naming conventions:

- we define our package **names in all lower case**
- package names are period-delimited
- names are also determined by **the company or organization that creates them**

To determine the package name based on an organization, we'll typically start by reversing the company URL. After that, the naming convention is defined by the company and may include division names and project names.

For example, to make a package out of *www.uno.edu*, let's reverse it:

```
edu.uno
```

We can then further define sub-packages of this, like *edu.uno.csci2120* or *edu.uno.csci2120.domain.*

4. Creating a Package - Directory Structure

Packages in Java correspond with a directory structure.

Each package and subpackage has its own directory. So, for the package edu.uno.csci2120, we should have a directory structure of edu → uno → csci2120.

Most IDE's will help with creating this directory structure based on our package names, so we don't have to create these by hand.

5. Using Package Members - Adding Class

Adding a class to a Package :

We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.

5. Using Package Members - Subpackages

Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement separated by dots (.)

Example: `package edu.uno.csci2120`

This approach allows us to group related classes into a package and their group-related package into a larger package. Store this package in a subdirectory names edu/uno/csci2120

A java file can have more than on class definition. In such cases, only one of the classes may be declared public & that class name with .java extension is the source file name. When a source file with more than one class definition is compiled, java created independent .class files for those classes.

5. Using Package Members - Subpackages

Subpackages:

Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

5. Using Package Members - Example

Let's start by defining a class TodoItem in a subpackage named domain:

```
package edu.uno.csci2120.scheduler;

public class TodoItem {
    private Long id;
    private String description;

    // standard getters and setters
}
```

5. Using Package Members → Imports

In order to use our *TodoItem* class from a class in another package, we need to import it. Once it's imported, we can access it by name.

We can import a single type from a package or use an asterisk to import all of the types in a package.

Let's import the entire *domain* subpackage:

```
import edu.uno.csci2120.scheduler.*;
```

Now, let's import only the *TodoItem* class:

```
import edu.uno.csci2120.scheduler.ToDoItem;
```

5. Using Package Members → Fully Qualified Name

Sometimes, we may be using two classes with the same name from different packages. For example, we might be using both *java.sql.Date* and *java.util.Date*. **When we run into naming conflicts, we need to use a fully qualified class name for at least one of the classes.**

Let's use *TodoItem* with a fully qualified name:

```
public class TodoList {
    private List<edu.uno.csci2120.scheduler.TodoItem> todoItems;

    public void addTodoItem(edu.uno.csci2120.scheduler.TodoItem todoItem) {
        if (todoItems == null) {
            todoItems = new ArrayList<edu.uno.csci2120.scheduler.TodoItem>();
        }todoItems.add(todoItem);
    }

    // standard getters and setters
}
```

6. Compiling with javac

When it's time to compile our packaged classes, we need to remember our directory structure. Starting in the source folder, we need to tell *javac* where to find our files.

We need to compile our *TodoItem* class first because our *TodoList* class depends on it.

Let's start by opening a command line or terminal and navigating to our source directory.

Now, let's compile our *edu.uno.csci2120.scheduler.TodoItem* class:

```
> javac edu/uno/csci2120/scheduler/TodoItem.java
```

6. Compiling with javac

If our class compiles cleanly, we'll see no error messages and a file *TodoItem.class* should appear in our *edu/uno/csci2120/scheduler* directory.

For types that reference types in other packages, we should use the *-classpath* flag to tell the *javac* command where to find the other compiled classes.

Now that our *TodoItem* class is compiled, we can compile our *TodoList* and *TodoApp* classes:

```
>javac -classpath . edu/uno/csci2120/scheduler/*.java
```

Again, we should see no error messages and we should find two class files in our *com/baeldung/packages* directory.

6. Compiling with javac

Let's run our application using the fully qualified name of our *TodoApp* class:

```
>java edu/uno/csci2120/scheduler/TodoApp
```

Our output should look like this:

7. Conclusion

- In this short lecture, we learned what a package is and why we should use them.

- We discussed naming conventions and how packages relate to the directory structure. We also saw how to create and use packages.

- Finally, we went over how to compile and run an application with packages using the *javac* and *java* commands.

END