# CSCI1620 Travel Agency – Supplemental Information

## Packages

As you may have noticed, our projects are beginning to be made up of a fair number of classes. It is common that complex programs are built out of numerous classes. It is helpful to organize our classes into related groups, similar to how you organize files on your home computer into folders/directories. This makes finding a specific class you are looking for in the project explorer much easier. In Java, this organization is achieved through what are called "Packages".

A class can be declared to belong to a package using the "package" keyword:

```
package examplepackage;
```

This package declaration must be the first executable line in the source file (comments above are ok!). By doing this, the class is now part of the package given, in the above example, the class would belong to the `examplepackage` package. The standard naming convention for packages is for them to be completely lowercase.

Within a project, packages are stored as a file structure. Let's say we have the class `ClassA` declared to be in the `examplepackage` package. Within the project's "src" folder, there would be a sub-folder named "examplepackage". Within that sub-folder is where the ClassA.java source code would be saved. Just like normal folders which can contain sub-folders, packages can have sub-packages. A sub-package can be declared by using a period to denote the sub-package inside another. Let's say we want the class `ClassB` to be declared in the package `subpackage`, which is a sub-package of `examplepackage`:

```
package examplepackage.subpackage;
```

Similar to the first example, a folder called "subpackage" will be placed in the "examplepackage" folder. The ClassB.java source will be in the "subpackage" folder.

Note how the period is used in the package declaration to represent a sub-package and sub-folder.

To create a package in Eclipse, right click on the project -> New -> Package. From here, you can name the package. When doing so, give the full name of the package you want to create. If creating a sub-package, but you still want the top-level package to appear in the project explorer, you will need to create the root package separately. In the example, I would create a package called `examplepackage` AND a package called `examplepackage.subpackage`. To create `ClassA`, I would right click on the `examplepackage` package when creating the class. To create `ClassB`, I would right click on the `examplepackage.subpackage` when creating the class.

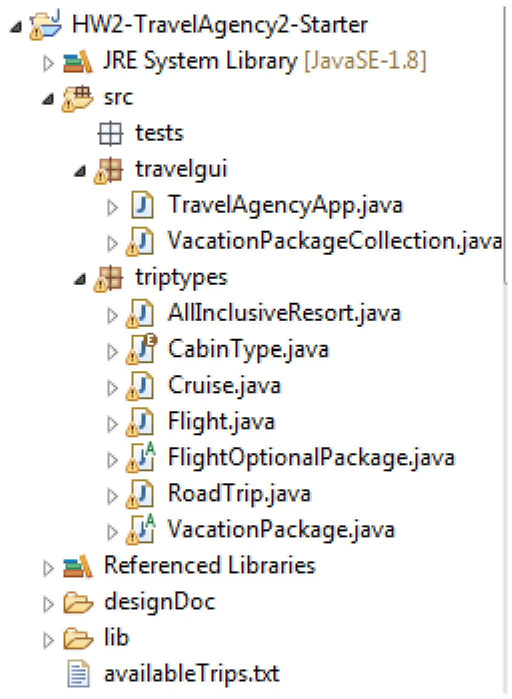
Besides organization, the other benefit of using packages is that it makes classes reusable! A class defined in a package can be imported into other classes in the project, or even into completely different projects! If you want to use a class that is not defined in the same package, it must be imported using the `import` keyword. This is the same `import` you've used to import classes like `Scanner`. I could import `ClassB` in a class that exists in a different package like this:

```
import examplepackage.subpackage.ClassB;
```

In UML, a package is denoted by a box around the classes it contains, with its name in the upper left. See the UML diagram for Assignment 2 for an example.

When working on Assignment 2, don't forget to import the classes that you need to use that are declared to belong to different package!

With the correct package structure, your project should look something like this in Eclipse's Project Explorer:



```
HW2-TravelAgency2-Starter
    JRE System Library [JavaSE-1.8]
    src
        tests
        travelgui
            TravelAgencyApp.java
            VacationPackageCollection.java
        triptypes
            AllInclusiveResort.java
            CabinType.java
            Cruise.java
            Flight.java
            FlightOptionalPackage.java
            RoadTrip.java
            VacationPackage.java
    Referenced Libraries
    designDoc
    lib
    availableTrips.txt
```

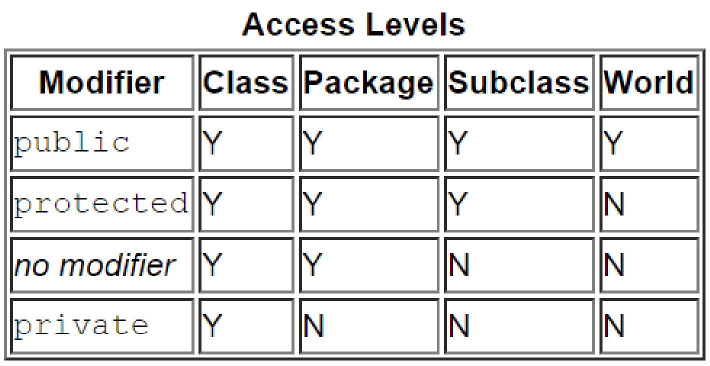
## Package and Protected Access

At this point, you are aware of the two extreme levels of access, `public` and `private`. There are two levels of access between: `package` and `protected`.

When a member of a class, data or method, is given `package` access, any other class that is declared to belong to the same package can access that member freely, as if it were `public`. If a member is created without explicitly giving an access level, it will default to `package` access.

Protected access relates to class inheritance. A subclass cannot access members of its superclass that are declared as `private`, even though it owns that member. When a member is declared as `protected`, subclasses will have the ability to access the member. Additionally, all other classes within the package can access it as well.

The following table displays all level of access and where they can be accessed from (*no modifier* is `package`):



Access Levels

| Modifier | Class | Package | Subclass | World |
|---|---|---|---|---|
| `public` | Y | Y | Y | Y |
| `protected` | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| `private` | Y | N | N | N |

Source: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

## WARNINGS FOR PACKAGE AND PROTECTED ACCESS:

Improper use of `package` and `protected` access can lead to poor Object-Oriented Design! When designing a class hierarchy, it may be tempting to declare all superclass variables as `protected` so subclasses can have access to them. However, this breaks the rules of encapsulation and information hiding!

Think of the `Employee/HourlyEmployee` example. In the `Employee` class, `firstName` was declared as `private`, meaning `HourlyEmployee` has a `firstName`, but cannot access it. If `firstName` were to be declared as `protected HourlyEmployee` would be able to access it directly. Sounds great! But there is now an issue. If `HourlyEmployee` can directly access `firstName`, it can bypass the control of that member that is enforced by `Employee`'s `setFirstName` method; specifically, if `firstName` is an empty `String`. This same rule would need to also be enforced in `HourlyEmployee`, leading to duplicated code, or methods within `HourlyEmployee` could ignore the rule altogether. Bad design!

As a general rule, data member access should be controlled solely by service methods, like getters and setters. Package and protected access are only to be used sparingly and only with good reason. Laziness and "I can't figure out how to make it work otherwise" are not good reasons! For assignments in this class, valid usages of `package` and `protected` will be provided to you in Javadocs. Any members you add **MUST** be `private`, or else design points will be deducted. Also know that the tests in Web-Cat or any black-box test exist in the "World", not within the packages you are creating. Things that are tested by Web-Cat must be public, do not change them to any other level of access.

## A primer on working with Calendar objects

Managing date and time information is a common programming task. In this assignment you'll need to track departure and arrival times for things like flights and cruises. Thankfully the standard Java library provides some classes to help us do this. We'll be using the `java.util.Calendar` class here; however, this class has a unique quirk that you may find strange at first. If you look carefully at the Javadoc, you may notice that this class is `abstract`, which means we can't call its constructor directly!

Thankfully, the class does provide a static method for retrieving a valid `Calendar` object configured in the current timezone. Once we have that, we need to set its information using a separate method call. Thus, if we want to create a Calendar object for Feb 20, 2019 at 11:59pm, we would say:

```
//create a valid object for the current day and time
Calendar deadline = Calendar.getInstance();

//update the object with a different day/time
deadline.set(2019, 1, 20, 23, 59);
```

Notice that the parameters to the `set` method are year, month, day, hour (24-hour clock), minute. Also you'll notice that for month we use 1 for February instead of 2. That's because months are 0-based in this API (with January being month 0 and December being month 11). You'll find other overloaded versions of `set` in the API documentation.

Now, if we want to get a pretty-printed view of this deadline object, we can use a second class (`java.text.SimpleDateFormat`) from the JavaAPI to do this:

```
SimpleDateFormat pretty = new SimpleDateFormat("HH:mm MM-dd-YYYY");
System.out.println(pretty.format(deadline.getTime()));
```

Changing what information to include or the order of fields is as simple as updating the format string provided to the `SimpleDateFormat` constructor!

**Remember** that to use these classes you'll want to import them at the top of your class definition!

For more details, see the full API documentation:

https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html

https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html