Exercise 16

Advanced packages

By the end of this exercise you will be able to

- Understand the idea of *package visibility* and why it is useful.
- Use `javadoc` to document a package.
- Use the `-d` option to keep `.html` files separate from `.java` files, and `.java` files separate from `.class` files.

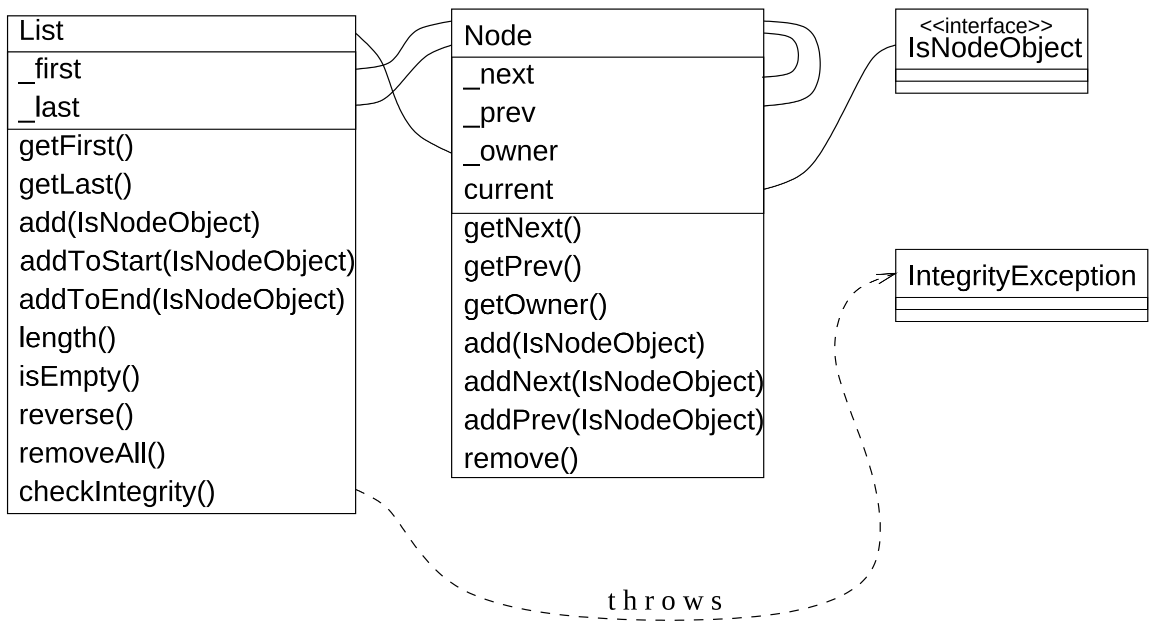
Introduction

From the previous exercise you will be familiar with how packages relate to files and directories and how you would store related classes in the same package. This exercise covers the idea of *package visibility* which is a new visibility modifier like `public`, `protected` and `private` that you will already be familiar with. Package visibility is a visibility level that sits in between `protected` and `private`. If we have defined a method or property `X` then here is a table showing how the visibility modifier for `X` relates to when `X` is visible.

	Public visibility	Protected visibility	Package visibility	Private visibility
In the same class as `X`	Yes	Yes	Yes	Yes
In the same package as `X`	Yes	Yes	Yes	No
In a subclass of `X`	Yes	Yes	No	No
Anywhere else	Yes	No	No	No

An example of package visibility

We will now consider an example that shows why the package visibility level is useful. The following UML diagram shows three classes and an interface that are used for implementing a system of doubly-linked lists. In the next few paragraphs, I will give a brief overview of how the system works in order to demonstrate how it benefits from package visibility.



The `List`, `Node` and `IntegrityException` classes and the `IsNodeObject` interface all reside within a package called `dlists`. The properties `_first`, `_last`, `_next`, `_prev` and `_owner` all have package visibility and are labelled with an underscore in front to emphasise this. The other methods and properties are all `public`.

The idea is for the user of the `dlists` system to use methods such as `add` and `remove` for manipulating doubly-linked lists rather than having to worry about the complex details of setting the properties such as `_next` and `_prev` to appropriate values. Here is a brief example showing the `dlists` system being used in practice:

```
List x = new List();

x.add(new StringNode("apple"));
x.add(new StringNode("banana"));
x.add(new StringNode("carrot"));

System.out.println(x); // prints: apple banana carrot

x.getFirst().getNext().remove(); // removes the second element

System.out.println(x); // prints: apple carrot

x.getLast().remove(); // removes the last element

System.out.println(x); // prints: apple
```

The `getFirst()`, `getLast()`, `getNext()`, `getPrev()` and `getOwner()` methods provide read-only access to the properties with underscores. It is vitally important that the user of `dlists` does not have write access to these properties, because the whole point of the system is to hide the details of setting these properties from the user. If the properties could be set by the user, then this would provide a way of undermining the structure of the linked list by setting the values of the properties incorrectly.

The requirement that it be impossible to set these properties from outside of the `dlists` package implies that the `public` visibility modifier cannot be used for these properties. Because methods of the `List` class need to access all the properties of the `Node` class and vice-versa, the `private` visibility modifier cannot be used either. Finally, since the two classes `List` and `Node` are not related by inheritance, the `protected` modifier cannot be used.

Package visibility is the only alternative to `public` visibility when several independent classes need to share access to methods, properties and constructors. By comparison, the C++ language has the `friend` keyword which has a similar use to Java's package visibility.

Unlike `public`, `protected` and `private`, package visibility is not indicated using a keyword, rather it is indicated by the absence of the keywords `public`, `protected` and `private`. Methods, properties, constructors, classes and interfaces can all have package visibility.

Questions

1. **Moving a class into a package.** Package visibility was designed to be used with packages. When the source file is not in any package, then package visibility means exactly the same thing as `public`. Often the Java programmer exploits this fact because it means that they can save some typing by using package visibility which has no keyword when they want something to be publicly available. The file `X.java` shows this trick being used. The method, property, constructor and the class are all given package visibility.

   The downside of this trick is that when the file is moved into a package, the method, property, constructor and class will need to have the `public` keyword added so they can be accessed from outside the package. This exercise takes you through the process of moving `X.java` into a package called `stuff`.

   (a) Without creating any package, fetch the files `X.java` and `Test.java` and compile them. Run `Test` and note what is output.

   (b) In the same directory that the previous two files are located, create a directory called `stuff` and move the file `X.java` into it.

   (c) Delete the file `X.class` because this contains the code for the `X` class before it was moved into a package. In general, you need to re-compile a class when you move it into a package.

   (d) In the file `X.java` add the following `package` declaration to the first line of the file:

   ```
   package stuff;
   ```

   (e) In the file `X.java` change the visibility of the class, method and constructor from package to `public` by adding the `public` keyword in front of the class, the method and the constructor. The property can be left with package visibility, because for the purposes of this question, it does not need to be accessed from outside the package.

   (f) In the file `Test.java`, add the following `import` statement to the first line of the file:

   ```
   import stuff.X;
   ```

   (g) Compile `X.java` and `Test.java` and then execute the `Test` class. The output of the program should be identical to what you saw in question 1a before you moved `X.java` into a package.

2. **Documenting a package.** This question will take you through the process of running the `javadoc` program on the files of the `dlists` package that was discussed earlier in this exercise.

   (a) Create a directory called `dlists` and fetch the following files into the `dlists` directory: `List.java`, `Node.java`, `IsNodeObject.java` and `IntegrityException.java`.

   (b) On the command line, `cd` to the parent directory of `dlists`.

(c) The following command will produce HTML files for the `dlists` package:

```
javadoc dlists
```

A problem with running `javadoc` in this way is that it causes your source code directories to be cluttered with HTML files. It is possible to tell `javadoc` to put the generated files in a different location like so:

```
mkdir -p ~/docs
javadoc -d ~/docs dlists
```

This creates a directory called `docs` in your home directory and tells `javadoc` to generate documentation in this directory.

(d) Load the file `index.html` into your web browser. If you used the `-d` option in the previous question, then the file will be located in the `docs` directory. Browse around the `List` and `Node` classes and notice that the properties with package visibility do not appear in the documentation. Usually this is the best arrangement but it is possible to get `javadoc` to show the extra properties. The `javadoc` manual is the place to look for these and other advanced features.

(e) If you have more than one package to document, then you should put the name of this package as an additional argument to `javadoc`. For example, the following command produces integrated documentation of the `dlists` and `stuff` packages that you have created so far:

```
javadoc dlists stuff
```

Of course, the `-d` option could be used in this case also.

3. **Keeping class files separate from source files.** The `-d` option that you used in the previous question can also be given to `javac` which causes it to put `.class` files in a separate directory from `.java` files. This question shows you how to compile and execute classes when the `-d` option is used.

(a) Fetch `ListTest.java`.

(b) Create a directory called `classes` in your home directory for storing `.class` files.

(c) Make sure that the current directory is appropriate and then compile `ListTest.java` and all of the source files of the `dlists` package using the following command. Note the use of the `-d` option:

```
javac -d ~/classes ListTest.java dlists/*.java
```

If you type `ls -R ~/classes`, you will see that `javac` has put created a subdirectory `dlists` of `classes` and put the classes that belong to the `dlists` package into this. The two classes that belong to no package have been put in the `classes` directory.

(d) When you use the `-d` option to compile your classes, you must execute your classes from the same directory as the one that you gave as an argument to `-d`. Here is how to execute the `ListTest` class:

```
cd ~/classes
java ListTest
```

Alternatively `~/classes` can be added to the class path if the `-classpath` option is being used.