Exercise 15

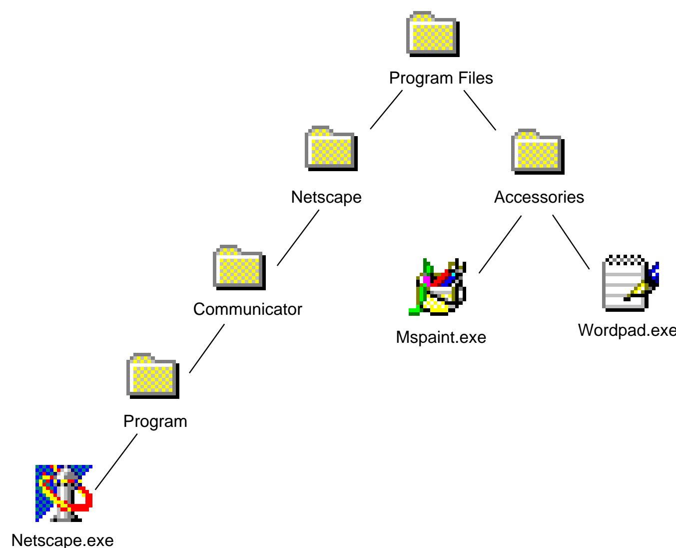
Introduction to packages

By the end of this exercise you will be able to

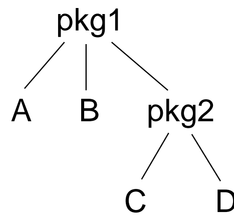
- Understand the ideas of *packages* and *subpackages* and how this is related to directories and subdirectories.
- Create a package and then compile and execute the classes within the package.
- Access classes from another package.

Introduction

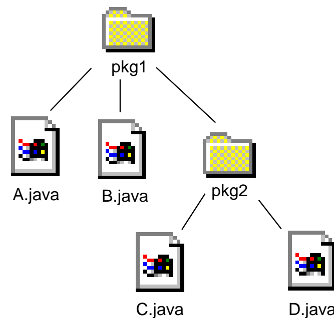
Packages are a tool for managing classes. You will be familiar with how files in a computer are stored in a tree system of directories so that files that are related to one another are stored in the same directory and how you can have more than one file of the same name, provided that they are stored in different directories. Here is a diagram of some of the files and directories on the `C:` drive of my computer.



Packages are a way of organising classes into a tree system like how directories are a way of organising files into a tree system. Classes that are related to one another are stored in the same package and you can have more than one class of the same name, provided that they exist in different packages. Here is a diagram showing some classes organised into packages.



The classes `A` and `B` belong to the package `pkg1`, and the classes `C` and `D` belong to the package `pkg1.pkg2`, which is a *subpackage* of the `pkg1` package. The source files for the classes in the above diagram must be stored in a directory structure that mirrors that arrangement of the classes within the packages.



Also the keyword `package` must be used as the first non-comment line in each of the four source files telling the compiler which package the source file is in. Here is what the file `A.java` might look like:

```
package pkg1;

public class A {
   public A() {
      System.out.println("class A's constructor called");
   }
   public void aMethod() {
      System.out.println("class A's aMethod called");
   }
}
```

Here is what the file `C.java` might look like:

```
package pkg1.pkg2;

public class C {
   public C() {
      System.out.println("class C's constructor called");
   }
   public void aMethod() {
      System.out.println("class C's aMethod called");
   }
}
```

Accessing a package

If you wish to access a class that is within another package from the package you are currently in, then there are three possible ways of doing this. The first way is to mention the package name in front of the class every time you mention the class. The source file `B.java` uses this technique:

```
package pkg1;

public class B {

   public static void main(String[] args) {

      java.util.Vector v = new java.util.Vector();

      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));

      java.util.Enumeration e = v.elements();

      while (e.hasMoreElements()) {
         Object o = e.nextElement();
         System.out.println(o);
      }

   }
}
```

The above example uses the `Vector` class and the `Enumeration` interface of the package `java.util`. Firstly a `Vector` object `v` is created, which is like an array that automatically resizes itself as elements are added, then some objects are added to it, and finally, the contents of `v` are printed to the screen. The `Enumeration` interface is a convenient way to iterate through the contents of a vector.

The disadvantage of needing to mention `java.util` every time that `Vector` is mentioned will be felt when vectors are used many times throughout the code. In such a case one would normally use the `import` facility to allow us to refer to `java.util.Vector` as simply `Vector`. Similarly we might want to refer to `java.util.Enumeration` as simply `Enumeration`. The `import` declarations just after the `package` declaration serves this purpose in the file `D.java` below:

```
package pkg1.pkg2;

import java.util.Vector;
import java.util.Enumeration;

public class D {

   public static void main(String[] args) {

      Vector v = new Vector();

      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));

      Enumeration e = v.elements();

      while (e.hasMoreElements()) {
         Object o = e.nextElement();
         System.out.println(o);
      }
```

```
    }
}
```

When you want to import several classes or interfaces from the same package, you might replace the several `import` statements at the top of the program with a single `import` statement:

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

This causes every class and interface of the `java.util` package to be imported. This allows us to directly refer to other useful classes such as `Stack` and `BitSet` of the same package. If they are not imported, then they can still be accessed, but they will need to have their package name mentioned every time the class name is mentioned like so: `java.util.Stack`, `java.util.BitSet`.

The questions of this exercise will take you through the process of putting the source files `A.java`, `B.java`, `C.java` and `D.java` into packages and then compiling and executing them.

Questions

1. **Creating a package.** Create a directory named `pkg1` and fetch the files `A.java` and `B.java` into it. Then create a directory called `pkg2` inside `pkg1` and fetch the files `C.java` and `D.java` into it.

2. **Compiling a package.** Generally, there is nothing special that needs to be done to compile the classes within a package that is different from how you compile a class that is not within a package. Simply run the `javac` program on all of the source files.

   On some occasions, like in question 5, the directory from which `javac` is invoked must be the parent directory of the package. On other occasions, the `-classpath` option must be used. See the online documentation for the `-classpath` option when compilation without `-classpath` fails.

3. **Executing a package.** As you will already know, any class that has a `main` method can be executed using the `java` program. If the class is within a package, then the directory that `java` is invoked from should be the parent directory of the package. Also the name of the package that the class is in must be stated before a dot followed by the package name as shown below:

   (a) On the command line, `cd` to the parent directory of `pkg1`.

   (b) Enter the following command:

   ```
   java pkg1.B
   ```

   to execute the `B` class. You should see the following output:

   ```
   1
   2
   3
   ```

   (c) Enter the following command:

   ```
   java pkg1.pkg2.D
   ```

   to execute the `D` class. You should see the same output as the previous question.

4. **Accessing a package.**

   (a) Create a file called `PackageAccess.java` in the parent directory of `pkg1`. Since the file is not in any package, there should be no `package` declaration at the top of the file.

(b) Type in the outer shell of the class definition for `PackageAccess` and a `main` method like so:

```
class PackageAccess
   public static void main(String[] args) {
   }
}
```

(c) Inside the `main` method, create an instance of the `A` class and assign it to a reference called `a`. Note that the `A` class is in a different package from the package you are currently in (no package), so there are three ways that this can be done.

HINT: The easiest way is to use the `import` statement at the top of the file.

(d) Call the method `aMethod` using the reference `a`.

(e) Compile `PackageAccess.java` and execute `PackageAccess` as you would normally for a file and a class that is not within a package.

5. **Accessing a subpackage.** To access a class that is in a subpackage of the current package, you follow the same steps as outlined in the previous question. This question takes you through the process of accessing the `C` class from the `B` class.

(a) After the `package` declaration in the file `B.java`, add the following `import` statement:

```
import pkg1.pkg2.C;
```

(b) At the end of `B`'s `main` method, add the following statements to construct a `C` object and call a `C` method:

```
C c = new C();
c.aMethod();
```

(c) Make sure that the current directory is the parent directory of `pkg1` and enter the following commands to re-compile `B.java` and execute the `B` class:

```
javac pkg1/B.java
java pkg1.B
```

NOTE: When you access a subpackage, Java is not smart enough to locate the subpackage on its own unless you give it a little extra help by having the current directory from which `javac` is invoked as the parent directory of the package. In some other cases the `-classpath` option must be used to help `javac` and `java` to locate classes.