CSE 143 Java

Class Packaging

Reading in N&H: Ch. 3, 4 (scattered)

Abstractions for Grouping Classes

- The class is the basic unit of modeling and program construction
- Most applications consist of more than one class
  - From a handful to hundreds of classes
- There should be ways to group related classes
- Three common levels of grouping
  - Within a file
  - Within a package
  - Within a project

Grouping Classes Within a .java file

- A single .java file may contain multiple classes
  BUT at most one public class
- If a file contains a public class...
  the file name *must* match the class name (case sensitive)
- Normal practice: put only very tightly related classes in the same file
  When in doubt, use more than one file
- Compiler will still generate a separate .class file for each class in a .java file
- A class can be nested (defined) inside another class
  Called "inner classes" – we will *not* use these in CSE143 at present

Packages

- Packages provide another way to group collections of related classes and interfaces
- A package defines a separate namespace to help avoid name conflicts
  - Can reuse common names in different packages (List, Set, ...)
- Provides a way of hiding classes needed to implement the package but that should not be used by client code

"Default" Package

- A type does not need to be in a named package
- There is an "anonymous" package for classes not placed in a specific package – you've been using this all along

Naming Packages and Types

- NB: for brevity, "type" here means "class or inface"
- Every class and interface has a *fully qualified* name: its **package name**, a ".", and its **type name**
  - java.util.ArrayList
  - java.awt.Rectangle
  - java.awt.geom.Ellipse2D
  - org.apache.xerces.parsers.SAXParser
- Each type also has a simple name
  - Color, ArrayList, Rectangle
- Can always refer to a type using its fully qualified name
- Can generally use import declarations to refer to types by their simple names
- Use **import** statements to make the package (and its type names) available in a program

Import Declarations (1)

- Can import a single type by giving its fully qualified name
  - import java.awt.Color;
- Can import all types in a package using the package name
  - import java.util.*;
- Have to import each package individually – can't import several in a single import declaration
- Example
  - import java.*;

  only imports top-level names in java.*
- To import, e.g., ArrayList, need to have (also)
  - import java.util.*

Import Declarations (2)

- An imported type can be referenced by its simple name, provided that reference is unique

```
import java.util.*;
ArrayList theList = new ArrayList( );
```

- Example of non-unique reference – both java.awt and uwcse.graphics contain a class Rectangle

```
import java.awt.*;
import uwcse.graphics.*;
Rectangle rect = new Rectangle(...);                  // error – ambiguous
java.awt.Rectangle r = new java.awt.Rectangle(...)    // ok; not ambiguous
```

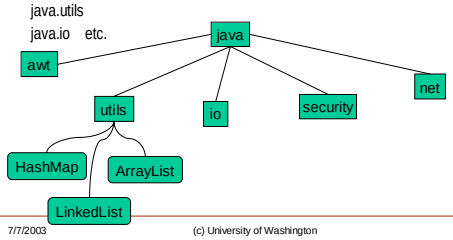
Creating Java Packages

- . Use the Java **package** statement
  - package statement must be first non-comment statement in the .java file
  - All .java files in a package must have identical package statements
- Packages vs directories
  - All .java files of a package *must* be in the same directory
  - The directory *must* have the same name as the package (case sensitive)
- Packages are often nested
  - dot notation: package, slash notation: director

Package Structure

- Packages can be nested in the usual directory structure
- This example contains the packages:

java
java.awt
java.utils
java.io etc.



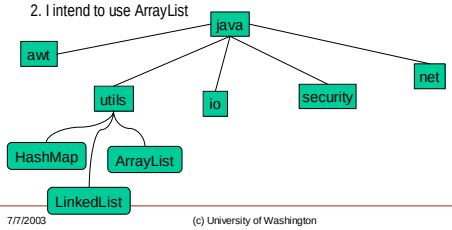
Packages and *import* Statement

- What does **import java.util.ArrayList;** mean?

1a. "There is a package named java which contains a package named util which contains a class named ArrayList (and I'm using that class)" OR

1b. "There is a directory named java which contains a directory named util which contains a class named ArrayList"

2. I intend to use ArrayList



Some Standard Packages

- The standard Java libraries contain thousands of classes grouped into dozens of packages. A few common ones:
  - java.lang – core classes; imported automatically everywhere, don't need an import declaration
    - includes Math, Integer, Double, Char, etc. – lots of useful things for standard types
  - java.util – collections, date/time, random number generators, etc.
  - java.io – input/output streams, files
  - java.net – network I/O, sockets, URLs
  - java.awt – original graphical user interface (GUI)
  - javax.swing – extension of awt, more sophisticated GUI

Java Standard Library Statistics

Version	#packages	# classes/interfaces
1.0	8	212
1.1	23	504
1.2	60	1781
1.3	77	2130
1.4	136	3020

Source: *The Java Developer's Almanac 1.4,* Patrick Chan

No, this will not be on the test

Class Paths

- The Java VM must know where to find classes at run-time
- The **class path** is a list of the directories which the JVM will search when looking for a class to load.
- Exercise: Open up DrJava or your favorite IDE. Find out what the class path is. Compare that with what your classmates report.
- Tricky thing: The class path does not contain the package, but the directory in which the package is found

Footnote: Using Internet Domains for Unique Names

- Java community convention: use reversed domain names as top-level package names

```
package com.sun.java.awt;
package edu.rice.cs.drjava;
```

- Overkill for simple projects, but a good idea if code is likely to be used by other organizations or groups

Projects

- Many IDE's have a "project" facility
- A way of grouping files needed for a particular application
  - May include .java files, .class files, directories, data files, images,etc.
  - May include entire packages
- Definition of “project” depends on a particular IDE
- Can a file be part of more than one project?
  - it depends
- Can a package might be used in more than one project?
  - it depends

Packages vs. Projects

- **package** is a Java concept
  - is a Java keyword
    - is a concept used in the **import** statement
  - is understood by the compiler
  - is understood by the JVM (Java Virtual Machine)
  - will be the same on all platforms, at least abstractly
- **project** is *not* a Java concept
  - each development environment has its own notion of what a project is
  - Often, a project is a directory
    - May be the same as a package
  - May instead be a non-package directory or something else
  - Some IDEs let you use a file or package in more than one project; some do not
  - Some environments have no project concept