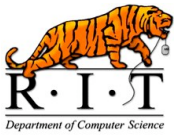


Inner classes in Java

First, a small sample problem

- You've been asked by your employer to develop a new linked list class
  - Remember that linked lists store information in "nodes", which the user never uses directly
- Some design questions arise:
  - Question #1: Should the user be able to use the Node class at all?
    - Answer: Probably not. At the least, it can confuse the issue when they're trying to figure out how to use the List class.
  - Question #2: How can we hide the node class away from the end user?

Introducing inner classes

- Inner classes are standard classes declared within the scope of a standard top-level class.
  - They are part of the class that contains them
  - They can be public/private/etc., just like data members and methods
- We've seen one example of an inner class already: Map.Entry

Types of inner classes

- There are four types of inner classes:
  - static member (a.k.a. nested top-level)
  - member
  - local
  - anonymous

Static member classes

- Declared static within a top-level class (sort of like a class-level member variable)
- They follow the same rules as standard classes:
  - private static classes cannot be seen outside the enclosing class
  - public static allows the class to be seen outside

So what?

- Why use a static member class?
  - If you have a type that is an essential part of an object's make-up, you can define that type as a part of the object's type

Some examples

- Think about the `Map` class:
  - an `Entry` (the key/value pair) is critical to the `Map`, so we make it an inner class
  - the user of the `Map` will want to be able to refer to the `Entry` objects (via `entrySet`), so it needs to be `public`
- Think about our new `List` class from earlier:
  - `Node` could be an inner class
  - the user won't be using `Node` objects, so the type can be `private`

Code snippet

```
public class DemoList
      implements List
   {
       private static class Node {
           public Node( Object value ) {
               this.value = value;
               next = prev = null;
           }
           public Object value;
           public Node next;
           public Node prev;
       }

       /** Keep track of start/end of the node list */
       Node head, tail;
       . . . .
```


Member classes

- A member class is a nested top-level class that is not declared `static`
- A member class has an additional `this` reference which refers to the enclosing class object
  - This reference is available via "`EnclosingClass.this`"
- Member objects are used to create data structures that need to know about the object they are contained in
- Member classes cannot declare static variables or methods, or have nested top-level classes

A simple example

- The `SimpleMemberClassDemo` class contains a member class named "`Member`".
  - Objects of this member class type have full access to the data members of the outer class that they're "associated with".
- [SimpleMemberClassDemo.java]

A better example

- Think about the iterators that we can get for a `Collection`/`List`
  - They need to know about the object they're "referring" to
  - They need to have full access to the collection's data so that it can be retrieved or removed
  - The user of our class should only be able to create these objects via the methods the class provides
- In `DemoList.java`:
  - `DemoListIterator` type
  - `iterator()` method

`this` Revisited

- To support member classes two extra kinds of expressions are provided:
  - `x = this.dataMember` is valid only if `dataMember` is an instance variable declared by the member class, not if `dataMember` belongs to the enclosing class
  - `x = EnclosingClass.this.dataMember` allows access to `dataMember` that belongs to the enclosing class
- Inner classes can be nested to any depth and the `this` mechanism can be used with nesting

new Revisited

- Member class objects can only be created if they have access to an enclosing class object
- This happens by default if the member class object is created by an instance method belonging to its enclosing class
- Otherwise it is possible to specify an enclosing class object using the `new` operator as follows:

```
EnclosingClass.MemberClass b =
   anEnclosingClassObject.new EnclosingClass.MemberClass();
```

Local classes

- A local class is a class declared within the scope of a compound statement, like a local variable
- A local class is a member class, but cannot include static variables, methods or classes. Additionally they cannot be declared `public`, `protected`, `private` or `static`
- A local class has the ability to access `final` variables and parameters inside the enclosing scope, as well as fields from the enclosing class.

Local Class Example

```
public class EnclosingClass {
  String name = "Local class example";

  public void aMethod( final int h, int w ) {
    int j = 20; final int k = 30;

    class LocalClass {
      public void aMethod() {
        System.out.println( h );
        // System.out.println( w ); ERROR w is not final
        // System.out.println( j ); ERROR j is not final
        System.out.println( k );
        // System.out.println( i ); ERROR i is not declared yet
        System.out.println( name); // normal member access
      }
    }

    LocalClass l = new LocalClass(); l.aMethod();
    final int i = 10;
  }

  public static void main() {
    EnclosingClass c = new EnclosingClass();
    c.aMethod( 10, 50 );
  }
}
```

Anonymous Class Syntax

- An anonymous class is defined as part of a `new` expression and *must* be a subclass or implement an interface:

```
ClassName var = new ClassName( argumentList )
                { classBody };
InterfaceName var2 = new InterfaceName()
                { classBody };
```

- The class body can define methods but cannot define any constructors
- The restrictions imposed on local classes also apply

Using Anonymous Classes

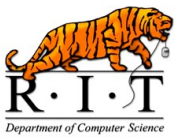
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MainProg {
  JFrame win;

  public MainProg( String title ) {
    win = new JFrame( title );

    win.addWindowListener(
      new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
          System.exit( 0 );
        }});
    . . . .
  }

  public static void main( String args[] ) {
       MainProg x = new MainProg( "Simple Example" );
  }
}
```



Packages in Java

A quick refresher....

Packages (a refresher)

- Java allows you to gather classes into logical groups
- Examples:
  - the various "core" classes, such as Object, String, etc., belong to the "java.lang" package.
  - the collections and other utilities belong to "java.util"
  - the AWT GUI classes belong to "java.awt"
  - the Swing GUI classes belong to "javax.swing"

Why packages?

- Packages allow us to:
  - use class names without worrying about conflicting type definitions
  - group code on disk so that it reflects logical organizations in programs
  - generate JavaDoc output that's easier to look through

Package naming

- Package naming conventions should be followed:
  - Package names start with your inverted Internet domain name
  - Beyond that, pick sub-package names that describe the problem domain
- Examples:
  - Software from RIT's CS department would belong to the "edu.rit.cs" package
  - Samples from this course might be in "edu.rit.cs.cs3"
  - Samples about inner classes might be in "edu.rit.cs.cs3.innerclasses"

Belonging to a package

- A class indicates that it is part of a package using the `package` statement
  - Example:
    ```
    package packageName;
    ```
  - This must be the first statement in a source file
  - If you don't specify the name of the package your class belongs to, it belongs to an unnamed (default) package.
- This tells the compiler that the "fully-qualified" name of the class is *packageName.className*

Using classes from packages

- You can provide the full name of the class
  ```
  java.util.List aList = new java.util.ArrayList();
  ```
- You can import the class into your code
  ```
  import java.util.ArrayList;          // at top of file
  . . . .
  java.util.List aList = new ArrayList();
  ```
- You can import the full package into your code
  ```
  import java.util.*;      // at top of file
  . . . .
  List aList = new ArrayList();
  ```

Other details

- Class packages can be stored in
  - directory hierarchies
  - ZIP archive files
  - JAR archive files
- Wherever the packages live, the base location (i.e., the root folder or the archive file) must be on the CLASSPATH

Mapping Packages to Files

- Package names map to directory names, and each directory contains all the `.class` files for a given package
  - The package `cs1.examples.stack` would map to `cs1/examples/stack`
  - The relative pathname is then appended to each entry in the CLASSPATH variable to create a full pathname