**Distribution Code**

package com.macmillan.nmeyers;

import java.lang.\*;

import java.lang.reflect.\*;

import java.util.\*;

import java.awt.\*;

class DumpClass

{

static class ClassMember implements Comparable

{

Member member = null;

ClassMember(Member m) { member = m; }

public String toString()

{ return member.toString(); }

public String getName()

{

String result = member.getName();

int pos = result.lastIndexOf('.');

if (pos != - ) result = result.substring(pos

+ );

return result;

}

// Implementation of compareTo: create an

ordering between

// ClassMember representations of members.

public int compareTo(Object m)

- -

{

if

(!member.getClass().equals(((ClassMember)m).member.

getClass()))

{

return

member.getClass().toString().compareTo(

((ClassMember)m).member.getClass().toString());

}

if

(member.getClass().equals(Constructor.class))

{

final Constructor constructor =

(Constructor)member;

final Constructor constructor =

(Constructor)((ClassMember)m).member;

int result =

constructor .getName().compareTo(

constructor .getName());

if (result != ) return result;

Class[] parm =

constructor .getParameterTypes();

Class[] parm =

constructor .getParameterTypes();

for (int i = ; i < parm .length && i <

parm .length; i++)

{

result =

parm [i].toString().compareTo(parm [i].

toString());

if (result != ) return result;

}

return parm .length - parm .length;

}

if (member.getClass().equals(Method.class))

{

final Method method = (Method)member;

final Method method =

(Method)((ClassMember)m).member;

int result = method .getName().compareTo(

method .getName());

if (result != ) return result;

Class[] parm =

method .getParameterTypes();

Class[] parm =

method .getParameterTypes();

for (int i = ; i < parm .length && i <

parm .length; i++)

{

result =

parm [i].toString().compareTo(parm [i].

toString());

if (result != ) return result;

}

return parm .length - parm .length;

}

if (member.getClass().equals(Field.class))

- -

return member.getName().compareTo(

((ClassMember)m).member.getName());

return member.toString().compareTo(

((ClassMember)m).member.toString());

}

}

// Our own implementation of a set, with some

filtering on the add()

// operation

static class MemberSet extends TreeSet

{

boolean showProtected = true;

boolean showPackage = false;

boolean showPrivate = false;

boolean showInaccessible = false;

String clsName = null;

String packageName = null;

MemberSet(Class cls, boolean f , boolean f ,

boolean f , boolean f )

{

showProtected = f ;

showPackage = f ;

showPrivate = f ;

showInaccessible = f ;

clsName = cls.getName();

int ppos = clsName.lastIndexOf(".");

packageName = (ppos >= ) ?

clsName.substring( , ppos) : "";

}

public boolean add(Class cls, ClassMember

element)

{

int modifier = element.member.getModifiers();

// Root out inaccessible members

if (!showInaccessible)

{

String cName = cls.getName();

int ppos = clsName.lastIndexOf(".");

String pName = (ppos >= ) ?

clsName.substring( , ppos) : "";

// Private ancestor members

if (Modifier.isPrivate(modifier) &&

!cName.equals(clsName) ¦¦

// Ancestor constructors

element.member.getClass().equals(Constructor.class) &&

!cName.equals(clsName) ¦¦

// Package-visible ancestors from

different packages

!Modifier.isPublic(modifier) &&

!Modifier.isProtected(modifier) &&

!pName.equals(packageName))

return false;

}

// This logic assumes relationships between

- -

permission

// levels that will always be true (e.g.

showPrivate->show

all).

if (Modifier.isPublic(modifier) ¦¦

showPrivate ¦¦

Modifier.isProtected(modifier) &&

showProtected ¦¦

!Modifier.isPrivate(modifier) &&

showPackage)

return super.add(element);

else return false;

}

}

// We don't need to link classes: provide a

classloader that skips

// that step.

static class MyClassLoader extends ClassLoader

{

public Class loadClass(String c) throws

ClassNotFoundException

{ return super.loadClass(c, false); }

}

static void usageMsg()

{

System.err.println("Usage: DumpClass [options]

<classes>");

System.err.println("\nOptions:");

System.err.println(" -public");

System.err.println(" -protected

(default)");

System.err.println(" -package");

System.err.println(" -private");

System.err.println(" -

suppress:{name,interfaces," +

"hierarchy,headings,keys,all}");

System.err.println(" -noancestors");

System.err.println(" -inaccessible");

System.exit( );

}

public static void main(String[] argv)

{

int argn;

boolean showProtected = true;

boolean showPackage = false;

boolean showPrivate = false;

boolean showName = true;

boolean showInterfaces = true;

boolean showHierarchy = true;

boolean showHeadings = true;

boolean showKeys = true;

boolean showAncestorMembers = true;

boolean showInaccessible = false;

for (argn = ;

argn < argv.length &&

- -

argv[argn].startsWith("-");

argn++)

{

if (argv[argn].equals("-public"))

showProtected = showPackage = showPrivate

= false;

else if (argv[argn].equals("-protected"))

{

showProtected = true;

showPackage = showPrivate = false;

}

else if (argv[argn].equals("-package"))

{

showProtected = showPackage = true;

showPrivate = false;

}

else if (argv[argn].equals("-private"))

showProtected = showPackage = showPrivate

= true;

else if (argv[argn].equals("-noancestors"))

showAncestorMembers = false;

else if (argv[argn].equals("-inaccessible"))

showInaccessible = true;

else if (argv[argn].startsWith("-suppress:"))

{

String args = argv[argn].substring( );

while (args.length() > )

{

int comma = args.indexOf(',');

String arg;

if (comma > )

{

arg = args.substring( , comma);

args = args.substring(comma + );

}

else

{

arg = args;

args = "";

}

if (arg.equals("name") ¦¦

arg.equals("all"))

showName = false;

if (arg.equals("interfaces") ¦¦

arg.equals("all"))

showInterfaces = false;

if (arg.equals("hierarchy") ¦¦

arg.equals("all"))

showHierarchy = false;

if (arg.equals("headings") ¦¦

arg.equals("all"))

showHeadings = false;

if (arg.equals("keys") ¦¦

arg.equals("all"))

showKeys = false;

- -

}

}

else

{

usageMsg();

}

}

MyClassLoader loader = new MyClassLoader();

// For each class requested

for (boolean firstClass = true;

argn < argv.length;

firstClass = false, argn++)

{

Class cls;

// Load the class

try

{ cls = loader.loadClass(argv[argn]); }

catch (ClassNotFoundException e)

{ System.err.println("Class " + argv[argn] +

" not found");

continue; }

// Build a set of members

MemberSet memberSet =

new MemberSet(cls, showProtected,

showPackage,

showPrivate,

showInaccessible);

// Step up the class hierarchy until we run

out. The indent

// controls indentation of classes in the

hierarchy chart, and

// is also used to ascertain when we're

operating on the

// requested class or an ancestor.

for (String indent = "";; indent += " ")

{

if (showName)

{

if (indent.equals(""))

{

// Do some processing specific to

the requested

class

if (!firstClass)

System.out.println("");

int modifiers =

cls.getModifiers();

String modString =

Modifier.toString(modifiers);

// Filter out weirdness in

handling of "interface"

// modifier

int pos = modString.indexOf("

interface");

- -

if (pos >= )

modString =

modString.substring( , pos) +

modString.substring(pos + );

if (modString.length() > )

System.out.print(modString +

" ");

}

if (showHierarchy ¦¦

indent.equals(""))

System.out.print(indent + cls);

// For the requested class, but not

superclasses, list

// the supported interfaces

if (indent.equals("") &&

showInterfaces)

{

Class[] interfaces;

interfaces = cls.getInterfaces();

if (interfaces != null &&

interfaces.

length > )

{

System.out.print(" implements

" +

interfaces[ ].getName());

for (int j = ; j <

interfaces.length; j++)

System.out.print(", " +

interfaces[j].getName());

}

}

if (showHierarchy ¦¦

indent.equals(""))

System.out.println("");

}

if (showAncestorMembers ¦¦

indent.equals(""))

{

// Build a list of methods for this

class

Method[] methods;

try

{ methods = cls.getDeclaredMethods();

}

catch (SecurityException e)

{ System.err.println("Security

exception

calling " +

"getDeclaredMethods() for " +

argv[argn]);

break; }

for (int j = ; j < methods.length;

j++)

- -

memberSet.add(cls, new

ClassMember(methods[j]));

// Build a list of constructors for

this class

Constructor[] constructors;

try

{ constructors =

cls.getDeclaredConstructors(); }

catch (SecurityException e)

{ System.err.println("Security

exception

calling " +

"getDeclaredConstructors()

for " +

argv[argn]);

break; }

for (int j = ; j <

constructors.length; j++)

memberSet.add(cls,

new

ClassMember(constructors[j]));

// Build a list of fields for this

class

Field[] fields;

try

{ fields = cls.getDeclaredFields(); }

catch (SecurityException e)

{ System.err.println("Security

exception

calling " +

"getDeclaredFields() for " +

argv[argn]);

break; }

for (int j = ; j < fields.length;

j++)

memberSet.add(cls, new

ClassMember(fields[j]));

}

// We're done when we run out of classes

to analyze

if (cls.equals(Object.class)) break;

cls = cls.getSuperclass();

if (cls == null) break;

}

// Output results

Class currentMemberType = null;

for (Iterator j = memberSet.iterator();

j.hasNext();)

{

ClassMember mm = (ClassMember)j.next();

if (showHeadings &&

- -

!mm.member.getClass().equals(currentMemberType))

{

currentMemberType =

mm.member.getClass();

if (currentMemberType ==

Constructor.class)

System.out.println("\nConstructors\n");

else if (currentMemberType ==

Field.class)

System.out.println("\nFields\n");

else

System.out.println("\nMethods\n");

}

if (showKeys)

System.out.print(mm.getName() + ": ");

System.out.println(mm.toString());

}

}

System.exit( );

}

}