elements of software construction

namespaces

Daniel Jackson
september 17, 2007
introduction
naming and namespaces

use of names
• is at the heart of programming
• name structure is large part of program design
• a balancing act

provide access with names
• allows flexible use of modules
• easy to add functionality

control access via names
• by making certain names unusable, can control access
• limit growth of complexity in programs
• prevent dependency on things that may change
hierarchical namespace
designing names

**desiderata**

- few symbols to name many things
- efficient lookup

**solution: hierarchy**

- form paths from sequences of symbols
- $S$ symbols, $k$-length paths give $S^K$ names

**example: URLs**

```
people.csail.mit.edu/dnj/talks.html
```

is really

```
edu/mit/csail/dnj/talks.html
```
interpreting with context

idea

- with hierarchical naming, can sometimes omit prefix
- prefix defines context, which may be implicit

example

- my phone number is 8-8471
- within MIT, this means 617-25/8-8471
overloading

interpretation of unqualified names
 • involves determining prefix
 • context may supply more than one
 • overloading: resolve by usage

possible cases
 • context declares a single prefix
   example: MIT phone extensions
 • context declares >1 prefix, ordered
   example: UNIX paths
 • context declares >1 prefix, resolve
   example: -3 - 4 resolved to /neg 3 /minus 4
   this is overloading
the Java namespace
elements, packages, imports

elements
- classes, interfaces, packages
- packages contain elements

placing a class in a package

```java
package myPackage;
class myClass {...}
// creates myPackage.myClass
```

using a class from another package

```java
package anotherPackage;
import myPackage.myClass;
// can now use myClass as if declared in anotherPackage
```
overloading

overloading in Java

\* only of method names
\* resolved by number and types of arguments
\* don’t confuse with overriding: overloading is completely static

example

\* given

```java
goal package java.util;
class ArrayList implements List {
    void add (int i, Object e) {...}
    boolean add (Object e) {...}
}
```

\* code fragment

```java
import java.util.ArrayList;
List a = new ArrayList ();
a.add (a); // resolves to second method
```
storage structure

where are files stored?

- each package containing files corresponds to a directory
- storage and namespace structures match
storage and namespace structures match? not quite

• need flexibility in where to root the hierarchy
• eg, my class `quoter.QuoteApp` is in the file

  `/usr/dnj/Filestore/Teaching/Fall05/code/quoter/QuoteApp.java`

classpath

• indicate where root of hierarchy is

  `classpath = /usr/dnj/Filestore/Teaching/Fall05/code`

• when several roots (eg, to accommodate libraries), class path
  becomes a Unix-style search path -- that is, a list of paths

  `classpath = .:/usr/bin:/usr/dnj/Filestore/Teaching/Fall05/code`

• classpaths are a nightmare, but fortunately Eclipse (mostly) hides them
the java library

some sample packages and classes

- java.
  - applet: Applet
  - awt: FileDialog
  - beans: Introspector
  - io: File
  - lang: Boolean
  - math: BigInteger
  - net: URL
  - nio: Buffer
  - rmi: FileDialog
  - security: CodeSigner
  - sql: SQLData
  - text: DateFormat
  - util: ArrayList
  - javax.sound.midi: MidiEvent
  - swing: JRadioButton
  - xml: parsers.DocumentBuilder

classes for making applets
user interface widgets
classes for making JavaBeans
files, streams, console
built-in datatypes
arbitrary precision arithmetic
network widgets
non-blocking i/o
remote method invocation
security for Java execution
for accessing relational DBs
language-independent text
collections framework
midi devices
platform-independent GUI
XML parsing
declarations and scope in Java
declarations & scope

declarations

‣ bind names to code, or to slots
‣ slots hold primitive values or object references

name use restricted

‣ to **scope** of declaration

by default, in Java

‣ all declarations have global scope
‣ except for local variables
scope of local variables

what is a local variable?
• declared within a method
• bound to slot only for duration of method call
• “stack allocated”

scope of local variable
• is code block in which declaration appears
• block is delineated by {}
block scope example

example

```java
public static void scopePlay () {
    int i = 0;
    while (i < 10) {
        int j = 0;
        while (j < 10) {
            int k = i * j;
            System.out.println (i + "*" + j + "=" + k);
            j++;
        }
        i++;
    }
}
```

exercise

\* suppose you made an error, and replaced `i++` by `j++` or `k++`
\* which error would the compiler catch?
nasty design flaw in Java

- try-catch forms a block
- try block encloses scope of declarations within
- so variable declared inside cannot be accessed after!

```java
try {  
    PrintStream s = new PrintStream (new FileOutputStream ("output.txt"));
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
s.println ("hello world"); // USE OUTSIDE SCOPE -- compile-time error
```

```java
PrintStream s = null;
try {  
    s = new PrintStream (new FileOutputStream ("output.txt"));
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
s.println ("hello world");
```
minimizing scope

principle: declare element in smallest scope it requires
• encourages decoupling of modules
• makes code easier to read
• within a file, avoids accidental access

examples
• use packages, and separate classes into subsystems
• use inner classes
• never use field when a local variable will do
• declare local variables at point of use, not all upfront
example: cut-and-paste error

bad cut-and-paste

```java
public static void scopePlay () {
    int i = 0;
    while (i < 10) System.out.println (i++);
    int j = 10;
    while (j < 20) System.out.println (i++);
}
```

fails to terminate

same error, but reduced scope

```java
public static void scopePlay () {
    {
        int i = 0;
        while (i < 10) System.out.println (i++);
    }
    {
        int j = 10;
        while (j < 20) System.out.println (i++);
    }
}
```

compile-time error
access control in Java
why global scope is bad

as noted earlier
• default scope for all elements is global
• so the method `p.q.r.m` can be accessed anywhere

but more access isn’t a good thing
• causes name clashes: overloading that can’t be resolved
• prevents local reasoning (don’t know who modifies my data)

solution: access modifiers
• for classes and interfaces: `public` or [default]
• for fields and methods and inner classes: `private`, `public`, `protected`, [default]
• default is always package private: only accessible in same package
• public = global, private = this class, protected = this package & subclasses
• so marking as protected *increases scope* [yes, this is weird]
public class Quoter {
  private URL url;
  private String open, ask;
  private int change;

  public Quoter (String symbol) throws MalformedURLException {
    url = new URL("http://quote.yahoo.com/d/quotes.csv?s=+symbol+&f=oap2");
  }

  public String getOpen () {return open;}
  public String getAsk () {return ask;}
  public int getChange () {return change;}

  public void obtainQuote () throws IOException {
    BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));
    String csv = in.readLine();
    in.close();
    StringTokenizer tokenizer = new StringTokenizer(csv, ",");
    open = tokenizer.nextToken();
    ask = tokenizer.nextToken();
    change = (int) (100 * (Float.valueOf(ask)-Float.valueOf(open)) / Float.valueOf(open));
  }
}
why are class and methods public?
  · so they can be accessed from another package

why are fields private?
  · to prevent outside writing
     different url for better service? then tokenizing may fail
     update open and ask? then change will be inconsistent
     (examples of rep invariant violation -- much more on this later)
  · to prevent outside reading
     suppose redefine change to be String not int
     client that uses change as int will no longer compile
general principles

make all fields private
  · and provide get and set methods

make all methods & classes designed for internal use private

make all other methods & classes public

don’t worry about protected
  · unless you’re building a framework
  · ... and you **really** need inheritance
review
summary

hierarchical naming

- standard in computing; flexible and powerful
- subspaces can be defined independently
- context, provided explicitly or implicitly
- overloading: resolve a name based on usage when context is not given

scope of an element

- syntactic region in which it can be used
- controlled in Java by location of decl, by {}, by access modifiers
- minimize scope to reduce errors and improve maintainability

read

- Bloch, Effective Java: Item 29: “Minimize the scope of local variables”