Programming Languages

Modules

Giuseppe Narzisi

Summer 2008
C++ namespaces

• What problem do namespaces solve?
  – suppose that you buy two different general-purpose class libraries, and each library has some features that you'd like to use. You include the headers for each class library:
  
  •  
  #include "vendor1.h"
  •  
  #include "vendor2.h"
It turns out that the headers have this in them:

```cpp
// vendor1.h

... various stuff ...

class String {

...;

};

// vendor2.h

... various stuff ...

class String {

...;

};
```
Problems with include directive

• This usage will trigger a compiler error:
  – class String is defined twice (compile-time clash).
  – there is the further problem of link-time clashes, since the two libraries contain identically-named symbols.
Solution...

// vendor1.h
... various stuff ...

namespace Vendor1 {
    class String {
        ...
    };
}

// vendor2.h
... various stuff ...

namespace Vendor2 {
    class String {
        ...
    };
}
For each unnamed namespace, the compiler generates a unique name, which differs from every other name in the program.

You might be asking yourself, *What's the use of this?*
Hiding names inside modules

• Before, this would be done using the static keyword.
• C++ Standard reads in section 7.3.1.1 Unnamed Spaces, paragraph 2:

  • "The use of the static keyword is deprecated when declaring objects in a namespace scope, the unnamed namespace provides a superior alternative."
ML Signature

- **Specification** of a structure
  - States the **types** which will be declared in the structure and gives the type information for the values and functions in the structure
- **Interface** which will **hide** some parts of the structure while allowing other parts to remain visible
  - it is possible for a structure to match a signature even though it declares more types and values than are required by the signature
Signature Ascription

• Imposes the requirement that a structure implement a signature

• There are two forms of ascription in ML:
  – *Transparent*, or *descriptive* ascription (:)
    • The structure is assigned the target signature augmented by propagating to the target the definitions of those types in the candidate.
  – *Opaque*, or *restrictive* ascription (:>)
    • The structure is assigned the target signature as is, without augmentation.

• Excessive use of transparent ascription impedes modular programming by exposing type information that would better be left abstract.
signature QUEUE =

sig

  type 'a queue

  exception Queue

  val empty : 'a queue

  val insert : 'a * 'a queue -> 'a queue

  val isEmpty : 'a queue -> bool

  val peek : 'a queue -> 'a

  val remove : 'a queue -> 'a * 'a queue

end
Structure QUEUE

```ml
structure TwoListQueue :> QUEUE =
struct
    type 'a queue = 'a list * 'a list
    exception Queue
    val empty = ([],[])
    fun insert (a,(ins,outs)) = (a::ins,outs)
    fun isEmpty ([],[]) = true
        | isEmpty _ = false
    fun peek ([],[]) = raise Queue
        | peek (ins,[]) = hd (rev ins)
        | peek (ins,a::outs) = a
    fun remove ([],[]) = raise Queue
        | remove (ins,[]) =
            let val newouts = rev ins
            in (hd newouts,([],tl newouts))
            end
        | remove (ins,a::outs) = (a,(ins,outs))
end
```
Opaque ascription (=>)

- States that any type components whose definitions are not provided in the signature (i.e., queue) should be treated as **abstract**
  - meaning that the definition of a queue as a pair of lists is not visible outside the module
There are two possible kinds of members in a package:

- **compilation units**
  - Java class or interface

- **subpackage**
  - contains compilation units and subpackages of its own
Hierarchical naming structure

- The *naming structure* for packages is hierarchical, with the hierarchy being separated by the ".":
  - java.lang
  - java.awt
  - java.awt.event

- However that the *relationship* between packages is not hierarchical.
  - There is no implicit relationship between the package java.awt and java.awt.event.
  - The compilation units in java.awt have no special access to the compilation units of java.awt.event, and vice versa.

- All packages should be thought of as being independent of one another.
  - Exception: all packages automatically import the predefined package java.lang.
How does Java compiler find files?

- The name of the class
- The names of imported packages (if any)
- The name of the current package

```java
import ListPkg.*;

public class Test {
    List L;
    ...
}
```

Assume that you are working in directory Javadir, which contains one file named Test.java
Closed and open scopes

- Module can be **open** or/and **closed** scopes
- **Closed scopes:**
  - Names must be explicitly imported
- **Open scopes:**
  - Modules that do not require imports

- **Note:** do not confuse this definition of open and close (from the textbook) with their definition used in the lectures:
  - the possibility to add new entities to a namespace/module/package after its initial definition.
Mutual Recursion in ML

- Testing whether a natural number is odd or even.
- Most obvious approach: test whether the number is congruent to “0 mod 2”.
- Solution using *mutual recursion*:

```ml
fun even 0 = true
    | even n = odd (n-1)
and odd 0 = false
    | odd n = even (n-1)
```

- We join their definitions using the keyword `and`. 