Object-Oriented Programming

CSCI-UA 0470-001
Class 3
Instructor: Randy Shepherd

Java & C++ Names

Name Management

- Classes, interfaces, methods and other entities have names by which they are referenced.

- During compilation these names need to be resolved and their meaning determined

- Programming languages deal with name management in different ways, but typically there is some notion of a ‘namespace’

Namespaces

- A namespace is a set of symbols that are used to organize objects of various kinds, so that these objects may be referred to by name
- Namespaces are commonly structured as hierarchies to allow reuse of names in different contexts
- Ex. file systems are namespaces that assign names to files
- Ex. databases have namespaces, schemas, tables, etc.

Namespaces

- As mentioned, programming languages typically support some mechanism for namespaces
- In Java, it is called ‘packages’
- In C++, it is called ‘namespaces’
- Strictly speaking packages in Java are more than simply namespaces.. What are some other features?

Java Packages

- To make types easier to find and use, to avoid naming conflicts and to control access, groups of related types are put into packages.
- Package correlate to directories on the file system, moreover the class org.example.Foo would be located in org/example/Foo.class
- Only one entity can exist with a particular name in a particular package.
- The package statement must be the first line in the source file.
    - ex. ‘package edu.nyu.oop;’
- There can be only one package statement in each source file, and it applies to all types in the file.
- If you do not use a package statement, your type ends up in an unnamed package.

Java Imports

- How to we ‘unpack’ them for use? import statements.
- To import a specific member of a package, put an import statement at the beginning of the file before any type definitions but after the package statement.
  - ex. ‘import xtc.lang.GNode;’
- To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.
  - ex. ‘import xtc.lang.*;’
- Now you are able to reference GNode by its name.
  - ex. ‘GNode node = GNode.create(“Foo”);’

Java Name Qualification

- Actually, imports are not strictly necessary!
- Languages often support simple and fully-qualified names.
- We could use the 'fully-qualified name'
  - ex. 'xtc.lang.GNode node = xtc.lang.GNode.create("Foo");'
- That's pretty ugly though. Who wants to do all that typing??
- Moreover, we can only utilize the simple names if we import the namepace of the type or if the referencing code is in the same package.

Java Packages Example

```java
// One package declaration, is the first line of the file
// Must match folder structure
package edu.nyu.oop.util;

// 0-to-many import statements immediately following
import xtc.tree.Node;

// Note that Visitor is referenced by its "fully-qualified name"
public abstract class RecursiveVisitor extends xtc.tree.Visitor {
    // ...whereas Node can be referenced by its "short name"
    public void visit(Node n) {
        for (Object o : n) {
            if (o instanceof Node) dispatch((Node) o);
        }
    }
}
```

C++ Namespaces

- Namespaces group named entities that otherwise would have global scope into narrower scopes.
- Allows organizing the elements of programs into different logical scopes referred to by names.
- Similar to packages in that only one entity can exist with a particular name in a particular namespace.

C++ Namespaces Example

```cpp
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

Differences to Java

- C++ Namespaces are unlike Java packages in that..
  - the folder structure has no relationship with namespace structure.
  - there is no relationship between a file and a package. You can have multiple namespaces defined in the same file.
  - namespaces do not provide access control.

C++ Using keyword

- The keyword ‘using’ imports a name into the the current region

- Like Java, you can include a specific entity from a namespace or all entities

Using keyword example

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

Java & C++ Sources

Source file management

- Need to find referenced names at compile time in order to make sure the classes etc. are used correctly
- How does the compiler know where to find definitions of classes?
- What steps are necessary prior to compilation?

Source Mgmt: Java

- Java relies on a convention
- One class per file
- Files arranged in directory hierarchy mirroring package names
- In some circumstances it is necessary to tell compiler root of source files hierarchy), this is done with the *-sourcepath* argument.
- This is abstracted away by our toolchain, however. In Sbt, we store all source files in <project-root>/src

Source File Mgmt: C++

- C++ relies on preprocessor (more on that in a minute)
- Dependencies need to be explicitly included; preprocessor actually includes text into source file before compilation
- Compiler takes files as arguments, if there are many files to be compiled you typically provide a ‘make’ file that lists the sources

C++ Source Files

- C++ header files:
  - Specify other included headers
  - Contain **declarations** of classes, methods, functions, globals, and constants
  - are named with the extension .h
- C++ implementation files
  - Contain **implementations** of classes, methods, functions
  - are named with the extension

C++ Compilation & Linking

- Implementation and header files are combined by and run through the preprocessor
- The resulting file is compiled into a .o unlinked file
- This is knows as ‘separate compilation’. The .o may be linked with other precompiled libraries so changes to one class would not necessarily require the re-compilation of the others
- Once complete, an executable binary is emitted named a.out by default

C++ Preprocessor and Directives

- The preprocessor modifies the text of the program
- It combines the sources prior to the compilation and linking steps
- It is possible to ‘program’ the preprocessor with directives
- Directives begin with a \`#\` and do not end in semicolons
- #pragma indicates a compiler-specific directive
- To see the output of preprocessor, run *g++ -E source_file*

Directive example

```
#include <iostream>
using namespace std;

#define PI 3.14159

int main ()
{

    cout << "Value of PI :" << PI << endl;

    return 0;
}
```

C++ include directive

- The #include directive includes the text of another file in the file that is being processed. This causes that file to be processed in turn.
- #include <foo> looks in the C++ standard header file path for foo
- #include <foo.h> looks in the C standard header path for the C header
- #include "foo" looks in the user's path
- Circular #includes may lead to an infinite recursion; the compiler stops at some point and prints an error
- To prevent these loops or conditionally exclude code, we have conditional directives, #IFDEF, IFNDEF and #ENDIF

Include guard example

```
#ifndef TEST_H
#define TEST_H

class Test
{
   public:
      void foo();
};

#endif
```

Java & C++ Binaries

Binary file management

- The system need to find code at runtime
- Statically linked code includes ALL dependencies in one binary
- Dynamically linked code resolves dependencies on demand
- We talked about this last week.

Binary File Mgmt: Java

- Java relies on dynamic linking
- Each class compiles into a single class file
- Again, directory hierarchy mirrors package names
- Java virtual machine can automatically find binaries, but we need to tell it where to look
- Java virtual machine can automatically find binaries, but we need to tell it where to look. classpath provided by CLASSPATH environment variable and/or -cp command line flag (also discussed this last week)

Jar Files

- Optionally, several class files may be grouped into a ‘jar’ file
- Jar file really is a zip file with some extra information. i.e., a compressed directory and file tree
- Short for Java ARchives
- Used for code distribution.

Binary File Mgmt: C++

- C++ relies on operating system
- Libraries loaded from library search path of operating system
- Code of several classes often grouped into one library
- All standard library code usually in one library

C++ Overview

Here be dragons

- C++ had a learning curve associated with it because you are “close to the metal”
- Compiler error messages are not always the best
- C++ compiler is much more permissive, its easy to write code that compiles but has errors at runtime. Pay attention to compiler warnings!
- C++ language is *vast*. Plenty of rope to hang yourself with.
- ..as a result we stick with a strict subset for this course. Basically anything necessary for our translation scheme, which we will review.

A quick note about C++ style

- C++ coders tend to be terse
- Names are usually lowercase, separated by underscores, “snake_case”
- In this class we will stick to Java style (camelCase), considering that our class favors clarity.
- Style tip: Scope-based naming rule-of-thumb, small scopes, small names, big scopes, big names!

Confused wolf says..
“enough jibber-jabber”