Composition of Classes

composition — an organized collection of components interacting to achieve a coherent, common behavior.

Why compose classes?

Permits a “Lego block” approach to design and implementation:
  Each object captures one reusable concept.
  Composition conveys design intent clearly.

Improves readability of code.

Promotes reuse of existing implementation components.

Simplifies propagation of change throughout a design or an implementation.

©2000-2002 McQuain WD & Keller BJ

Composition by Association

Association (acquaintance)

Example: a database object may be associated with a file stream object.

The database object is “acquainted” with the file stream and may use its public interface to accomplish certain tasks.

Acquaintance may be one-way or two-way.

Association is managed by having a “handle” on the other object.

Associated objects have independent existence (as opposed to one being a sub-part of the other).

Objects have "meaning" apart from their association.

Sometimes referred to as the “knows-a” relationship.

©2000-2002 McQuain WD & Keller BJ

Establishing an Association

The objects that will be involved are created independently by the driver code.

The driver code "introduces the objects" by passing one object, or its address, to the other object which stores a pointer to maintain the association.

Association is generally established dynamically (at run-time), although the design of one, or both, of the classes must make a provision for establishing and maintaining the association.

An association can also be implemented as an object, but that is not required for most abstractions.

©2000-2002 McQuain WD & Keller BJ

A Simple Association


```
class DisplayableNumber {
protected:
   int      Count;
   ostream* Out;
public:
   DisplayableNumber(int InitCount = 0, ostream* Where = &cout);
   void ShowIn(ostream* setOut);
   void Show() const;
   void Reset(int newValue);
   int  Value() const;
   bool operator==(const DisplayableNumber& Other);
   ~DisplayableNumber();
};
```

```
void DisplayableNumber::ShowIn(ostream* setOut) {
   Out = setOut;
}

void DisplayableNumber::Show() const {
   *Out << Count << endl;
}
```


©2000-2002 McQuain WD & Keller BJ

Establishing the Association

Here, the `DisplayableNumber` has the responsibility for maintaining the association with a particular output stream, and gives the user the ability to set or change the targeted stream as desired.

```
DisplayableNumber Counter;            // default association

ofstream Out("Scores.data");
DisplayableNumber Score(0, &Out);     // explicit association

Score.Reset(Score.Value() + 1);
Score.Show();

Counter.ShowIn(&Out);                 // reset association
```

Note the independence of the `DisplayableNumber` object and the associated stream objects.

©2000-2002 McQuain WD & Keller BJ

Potential Problems

Consider:

```
DisplayableNumber Counter;          // default association

Counter.ShowIn(NULL);               // reset association
Counter.Show();                     // really bad. . .
```

Here, the invocation of `Show()` on `Counter` should result in a runtime violation, as a `NULL` pointer will be dereferenced.

The issues that usually arise when pointers are used may arise with an association between objects. It is generally going to be the responsibility of the "invoking" object to be sure that an association actually exists before attempting to exploit it.

Here, `Counter` fails to do that.

©2000-2002 McQuain WD & Keller BJ

Static *vs.* Dynamic Association

<u>static</u> association cannot change. It is usually established via an object constructor, and there is no mutator function that would allow it to be modified.

<u>dynamic</u> association uses method(s) that allow changing who is associated with the object. Initial association may still be established at construction.

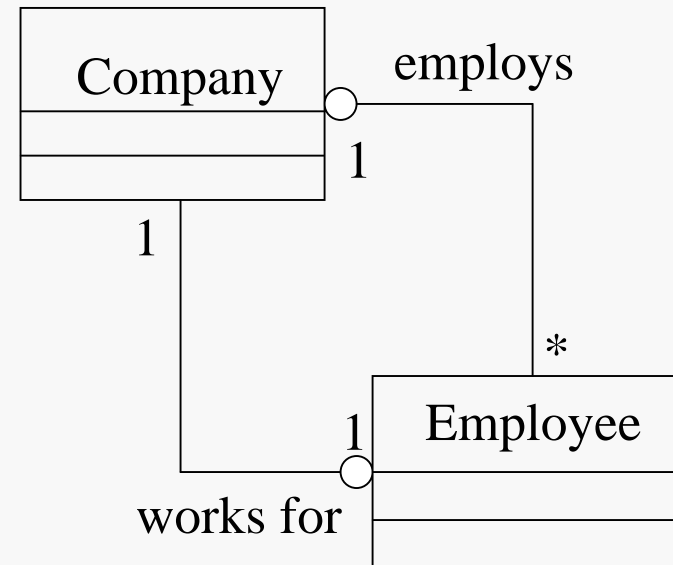
Consider:

- the relationship between the `CirculationDesk` and the `Catalog` objects in the library system
- the relationship between a `DisplayableNumber` object and a stream

©2000-2002 McQuain WD & Keller BJ

Association in Class Diagrams

In a class diagram, an association is represented by a line with a circle at the boundary of the object storing the association:



The association arrow may be labeled with a brief description of the logical relationship the association represents.

The head and terminus of the association arrow may be labeled with integers or an asterisk, indicating the number of objects involved in each side of the association.

We will call these the multiplicity values.

©2000-2002 McQuain WD & Keller BJ

Cardinalities

Not every relationship is one-to-one.

Specify cardinalities of relationships by numbers/symbols at ends of association arrows.

Possibilities:

1:1	one to one
1:2	one to two
1:0…n	one to from 0 to n
1:*	one to any number (including none)
2:2	two to two
m:n	m to n
:	any number to any number

©2000-2002 McQuain WD & Keller BJ

Using `this` for Association

In some situations, one object may "register" itself with another, establishing an association dynamically.

How can an object provide a pointer to itself??

```
class Passenger {
private:
   . . .
   int Floor;     // passenger knows what floor he's on
public:
   . . .
   void Board(Elevator* pE) const;
   . . .
};

void Passenger::Board(Elevator* pE) const {
   if ( pE->onFloor() == Floor ) // elevator can say what floor
                                 //   it's on
      pE->addPassenger(this);    // passenger registers himself
}
```

 ©2000-2002 McQuain WD & Keller BJ

Association Compilation

Assume we have a class A that contains a pointer to class B and that class B contains a pointer to class A. If we declared them as follows:

```
//FILE: A.h
#ifndef CLASSA_H
#define CLASSA_H

#include "B.h"
class A {
     private:
       B* ptr2B;
     public:
       A();
       A(B* pb);
   };
#endif
```

```
//FILE: B.h
#ifndef CLASSB_H
#define CLASSB_H

#include "A.h"
class B {
     private:
            A* ptr2A;
     public:
          B();
            B(A* pa);
   };
#endif
```


©2000-2002 McQuain WD & Keller BJ

Compilation Problems

When the following code.cpp files are compiled this results in the missing storage-class or type specifier error in the private declaration of class B (i.e. the "A* ptr2A;").

```
//FILE: A.cpp
#include <new>
using namespace std;

#include "A.h"

A::A()       {ptr2B = NULL;}
A::A(B* pb){ptr2B = pb;}
```

```
//FILE: B.cpp
#include <new>
using namespace std;

#include "B.h"

B::B()       {ptr2A = NULL;}
B::B(A* pa){ptr2A = pa;}
```

In compiling A.cpp, the following inclusions result:

```
//FILE: A.cpp
#include <new>

#include "A.h"
```

1. <new> is expanded by preprocessor (no problem)
2. A.h is expanded which itself includes `"B.h"`
3. B.h is then expanded which re-includes `"A.h"`
4. The re-inclusion of `"A.h"` fails due to the directive `#ifndef CLASSA_H`

Thus A has not been defined when the compiler reaches the 2nd Class B constructor.

©2000-2002 McQuain WD & Keller BJ

Class Forward Declarations

When an indirect header file dependency problem arises use class forward declarations to resolve the inclusion problems.

```
//FILE: A.h
#ifndef CLASSA_H
#define CLASSA_H

//#include "B.h"
class B; //Forward

class A {
     private:
       B* ptr2B;
     public:
       A();
       A(B* pb);
   };
#endif
```

```
//FILE: B.h
#ifndef CLASSB_H
#define CLASSB_H

//#include "A.h"
class A;  //Forward

class B {
     private:
             A* ptr2A;
     public:
           B();
             B(A* pa);
   };
#endif
```


©2000-2002 McQuain WD & Keller BJ

Header Includes

Header files are included in the code.cpp files where they are needed and will not cause indirect inclusion problems:

```
//FILE: A.cpp
#include <new>
using namespace std;

#include "A.h"
#include "B.h"

A::A()      {ptr2B = NULL;}
A::A(B* pb){ptr2B = pb;}
```

```
//FILE: B.cpp
#include <new>
using namespace std;

#include "B.h"
#include "A.h"

B::B()      {ptr2A = NULL;}
B::B(A* pa){ptr2A = pa;}
```

 ©2000-2002 McQuain WD & Keller BJ