Lecture #12 | Polymorphism: templates

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Previously in Object-Oriented Programming

- Polymorphism: virtual functions
  - Functions with the same name may behave differently, depending on which types (i.e., classes) they are associated with

Today’s topic

- Templates

Templates

- A *template* is a class or a function that we parameterize with a set of types or values
- We represent general ideas from which we can generate specific classes or functions by providing types (e.g., int, double, or user-defined class) as parameters
- Syntax: both `class` and `typename` is valid, but the latter is recommended

```
template<typename Type> function_declaration;
template<class Type> function_declaration;

template<typename Type> class_declaration;
template<class Type> class_declaration;
```

Example: function templates

```
template<typename T>
T add(const T x, const T y) {
    return x + y;
}

int main() {
    cout << add<int>(42, 23) << endl;
    cout << add<double>(3.14, 1.68) << endl;
    cout << add(3.14, 1.68) << endl;
}
```

- By using function templates, we do not have to implement the same functions for different types
- When using function template, you may omit template argument if the required type can be unambiguously deduced

Example: class templates

```
template<typename T>
class Point {
    T x;
    T y;
public:
    Point(T xx=0, T yy=0) : x(xx), y(yy) {}
    T getX() { return x; }
    T getY() { return y; }
};
int main()
{
    Point<double> pt_d {1.2, 3.4};
    cout << pt_d.getX() << endl;
    Point<int> pt_i {1, 2};
    cout << pt_i.getX() << endl;
}
```

- By using class templates, we do not have to implement the same classes with different types

Example: class templates with more than one types

```
template<typename T1, typename T2>
class Pair {
public:
    T1 first;
    T2 second;
};

int main()
{
    Pair<int, double> pair {42, 3.14};
    cout << pair.first << " ”
         << pair.second << endl;
}
```

- You may have as many template arguments, i.e., template types

Example: class template with values

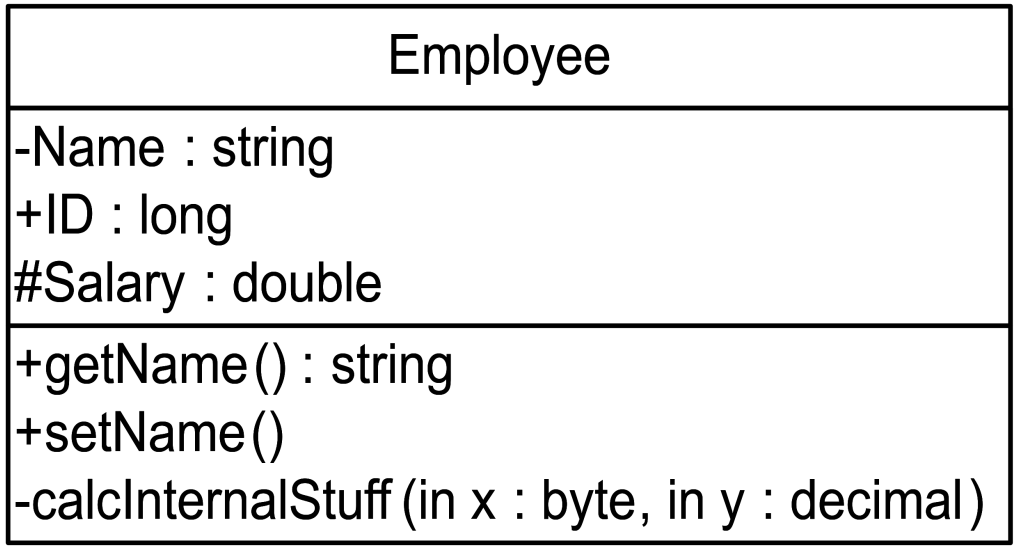
```
template<typename T, int dim>
class PointND {
    T* coordinates;
public:
    PointND() { coordinates = new T[dim]; }
    ~PointND() { delete[] coordinates; }
    int getDimension() { return dim; }
};
int main() {
    PointND<double, 2> p2;
    PointND<double, 3> p3;

    cout << p2.getDimension() << endl;
    cout << p3.getDimension() << endl;
}
```

Is-a v.s. Has-a

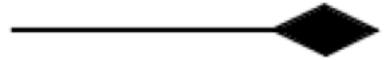
- Two ways we can describe some class A depending on some other class B
  - Every A object has a B object. For instance, every Vehicle has a string object (called license or name)
  - Every instance of A is a B instance. For instance, every Car is a Vehicle, as well
- ***Inheritance*** allows us to define “is-a” relationship, but it should not be used to implement “has-a” relationships
- Sometime it is not clear whether to use “is-a” or “has-a” relationship

UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e., member variables)
  - Class operations (i.e., member functions)
- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: underlined
- Abstract class: name in italics

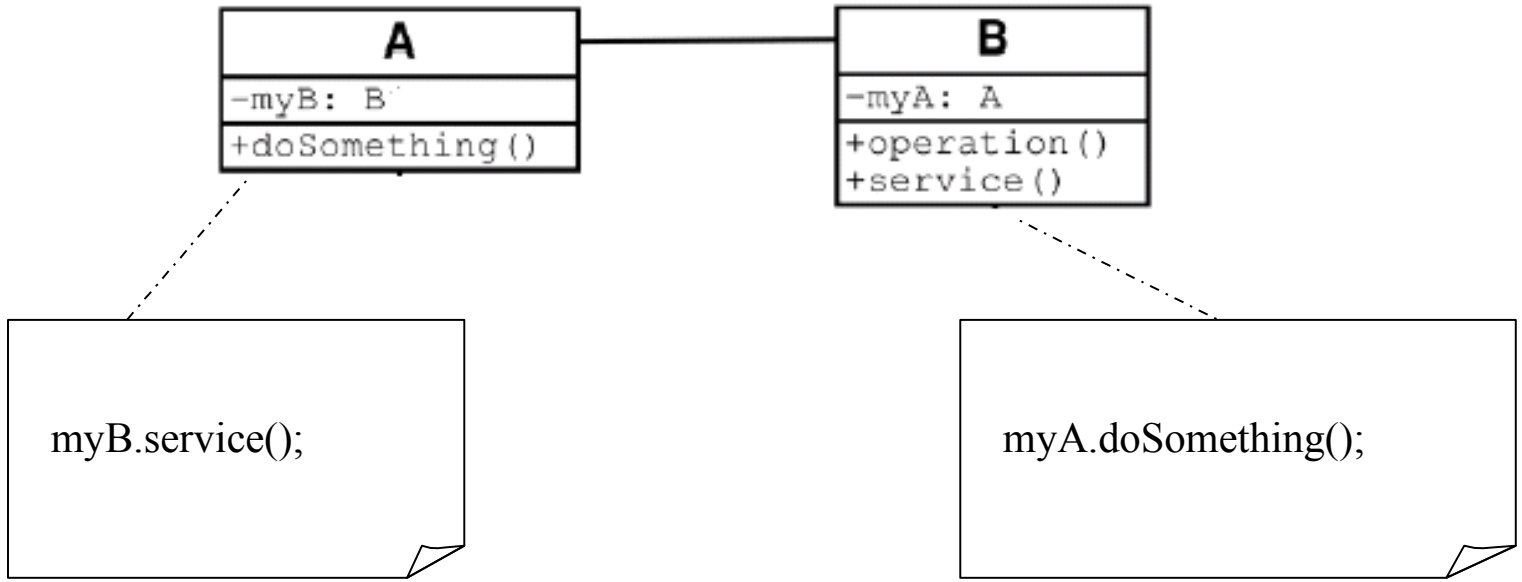


UML Class relations

- **Association**: a straight line or arrow 
  - A relationship between instances of two classes, where one class must know about the other to do its work, e.g., client communicates to server
- **Aggregation**: an empty diamond on the side of the collection 
  - An association where one class belongs to a collection, e.g., instructor part of faculty
- **Composition**: a solid diamond on the side of the collection 
  - Strong form of aggregation
  - Lifetime control; components cannot exist without the aggregate
- **Inheritance**: a triangle pointing to superclass 
  - An inheritance link indicating one class a superclass relationship, e.g., bird is part of mammal

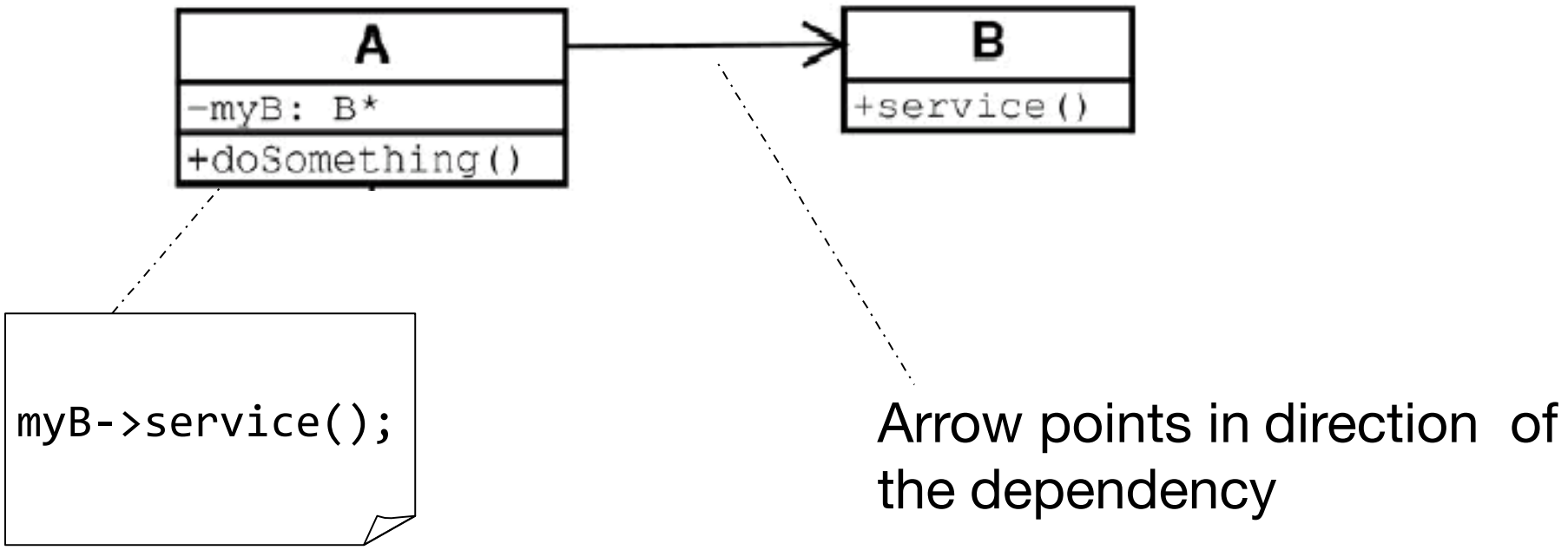
Binary Association

- Both entities “know about” each other



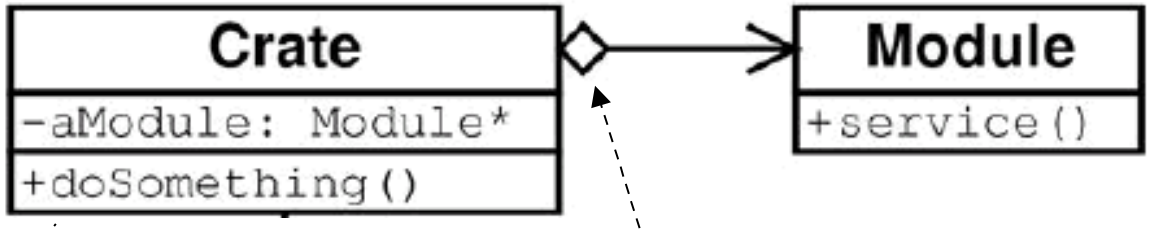
Unary Association

- A knows about B, but B knows nothing about A



Aggregation

- Aggregation is an association with a “collection-member” relationship



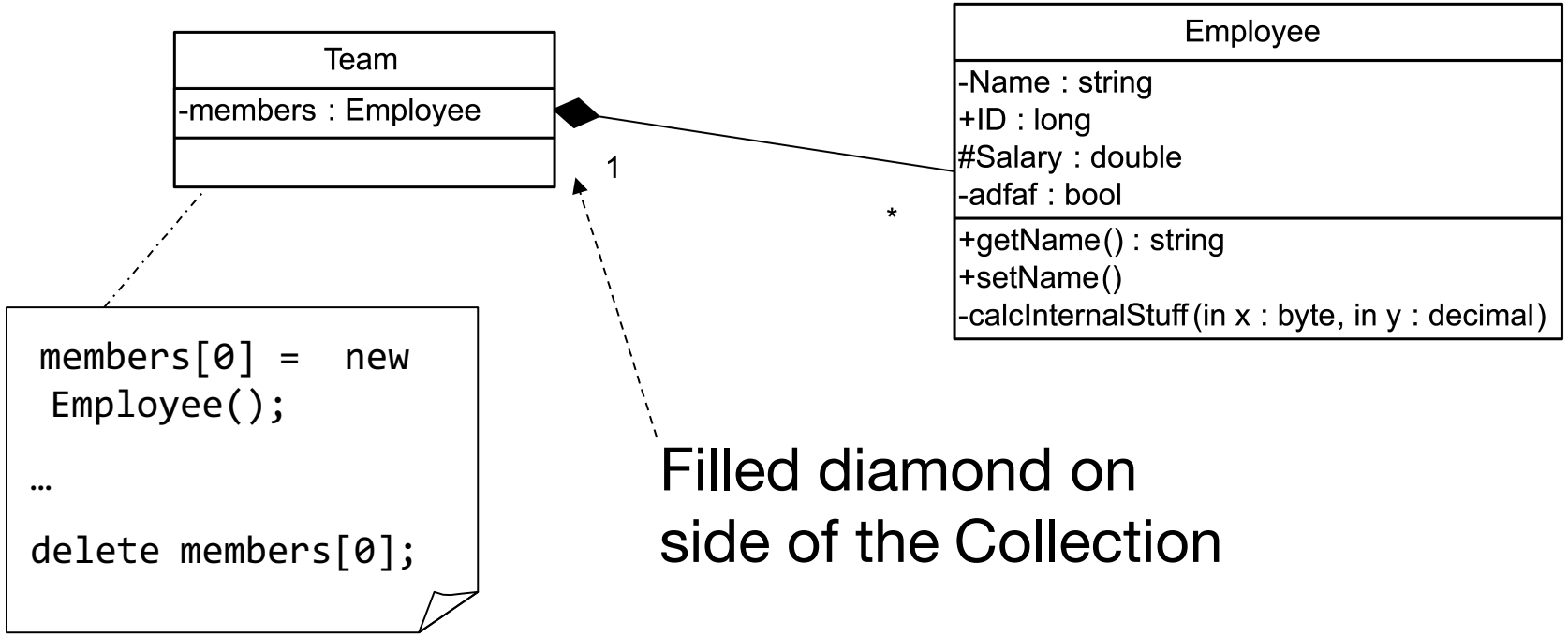

```
void doSomething() {
  aModule->service();
}
```

Hollow diamond on
the Collection side

No sole ownership implied

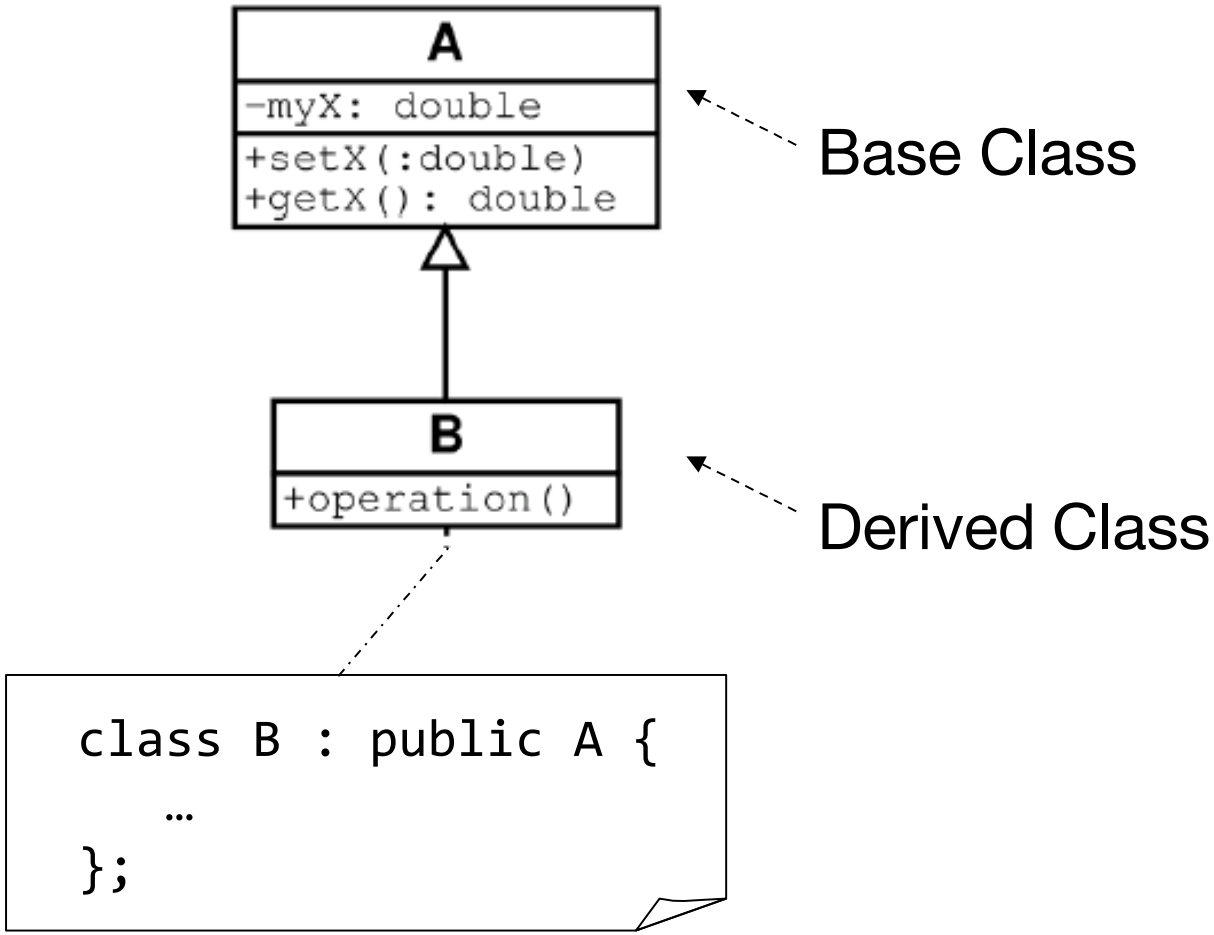
Composition

- Composition is aggregation with
  - The whole-part relationship
  - Lifetime control (owner controls construction & destruction)



Inheritance

- Standard concept of inheritance

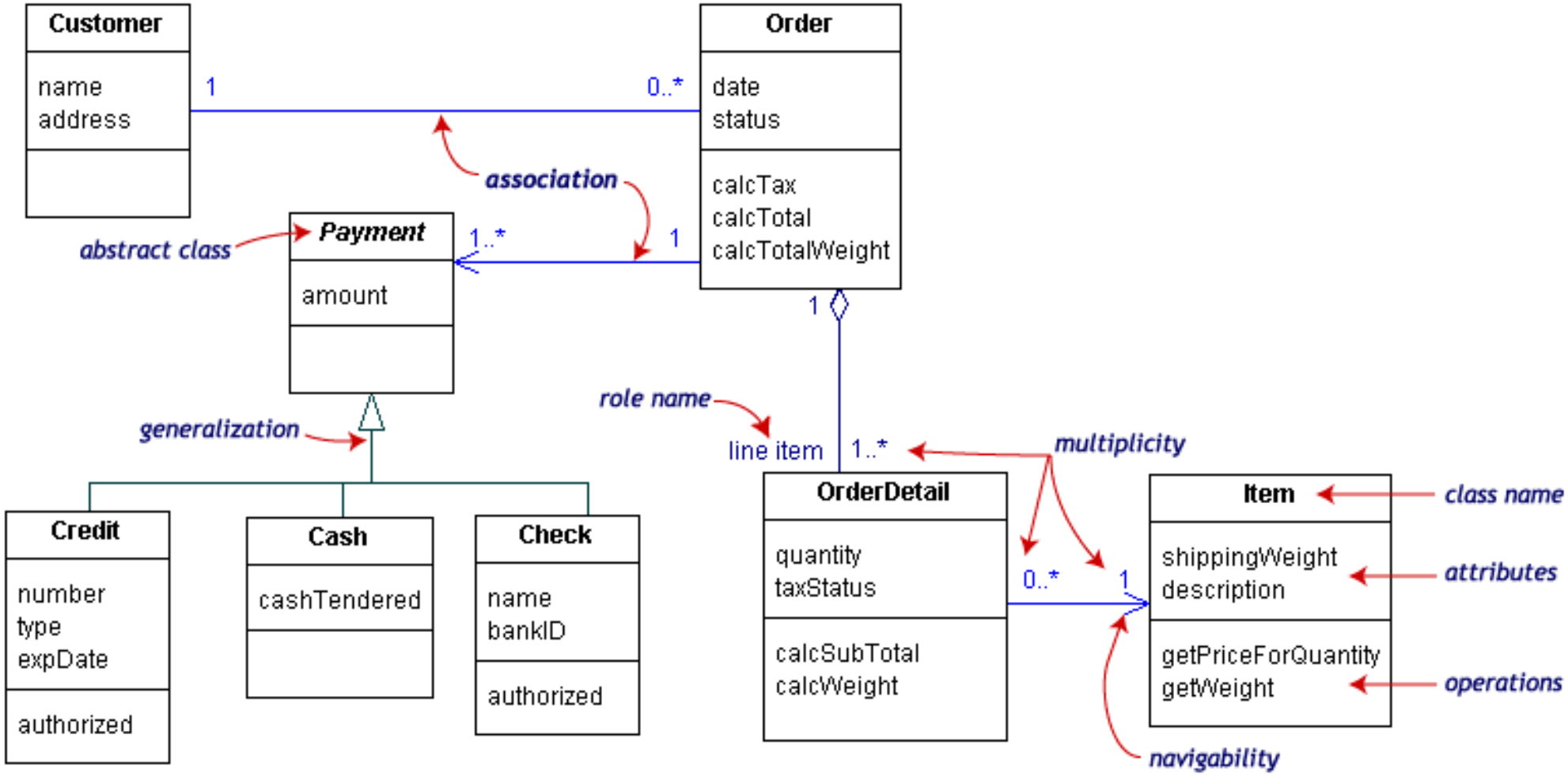


UML Multiplicities

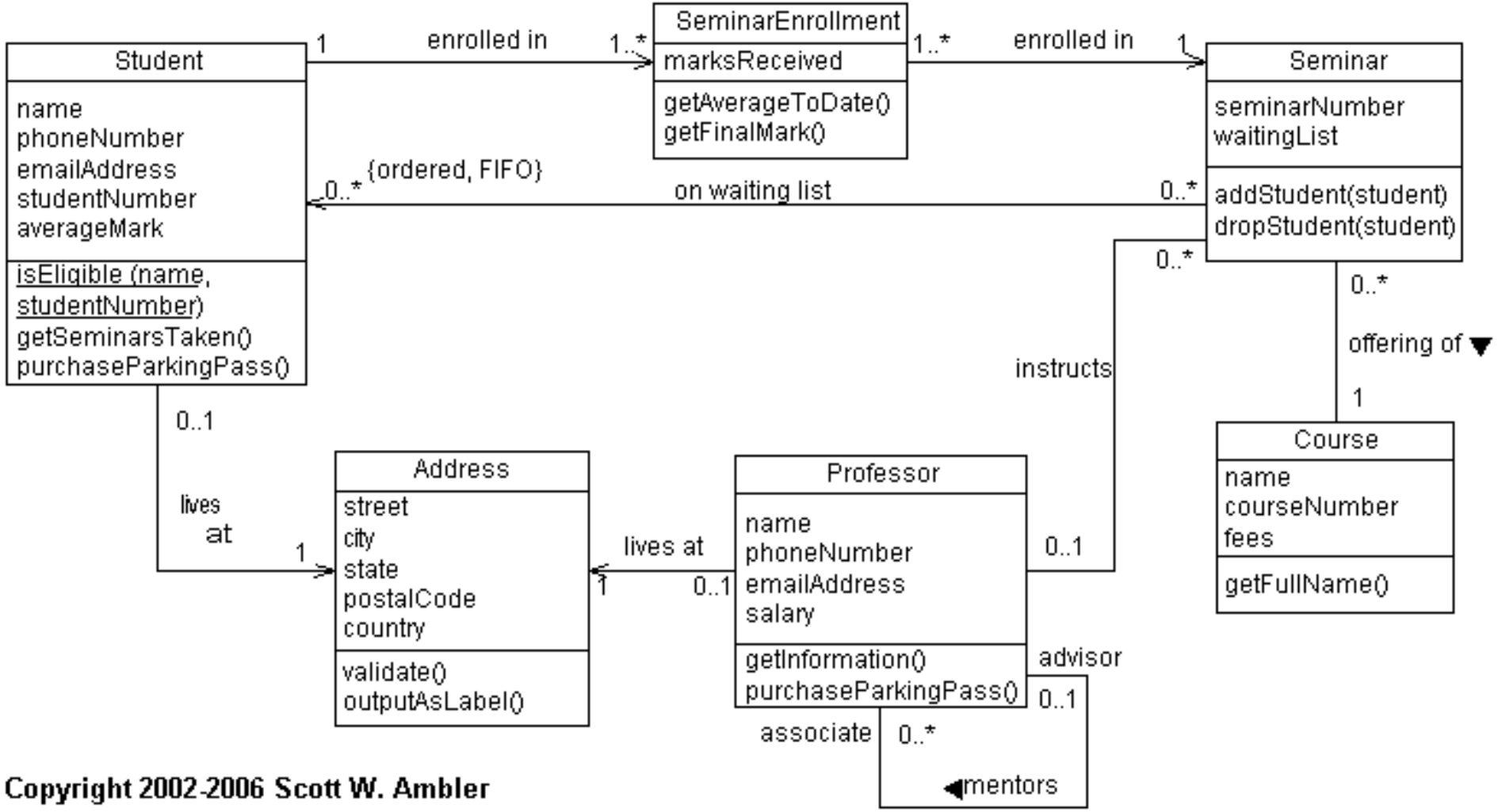
- Links on associations to specify more details about the relationship

Multiplicities	Meaning
n..m	n to m instances
*	no limit on the number of instances (including none)
1	exactly one instance
1..*	at least one instance

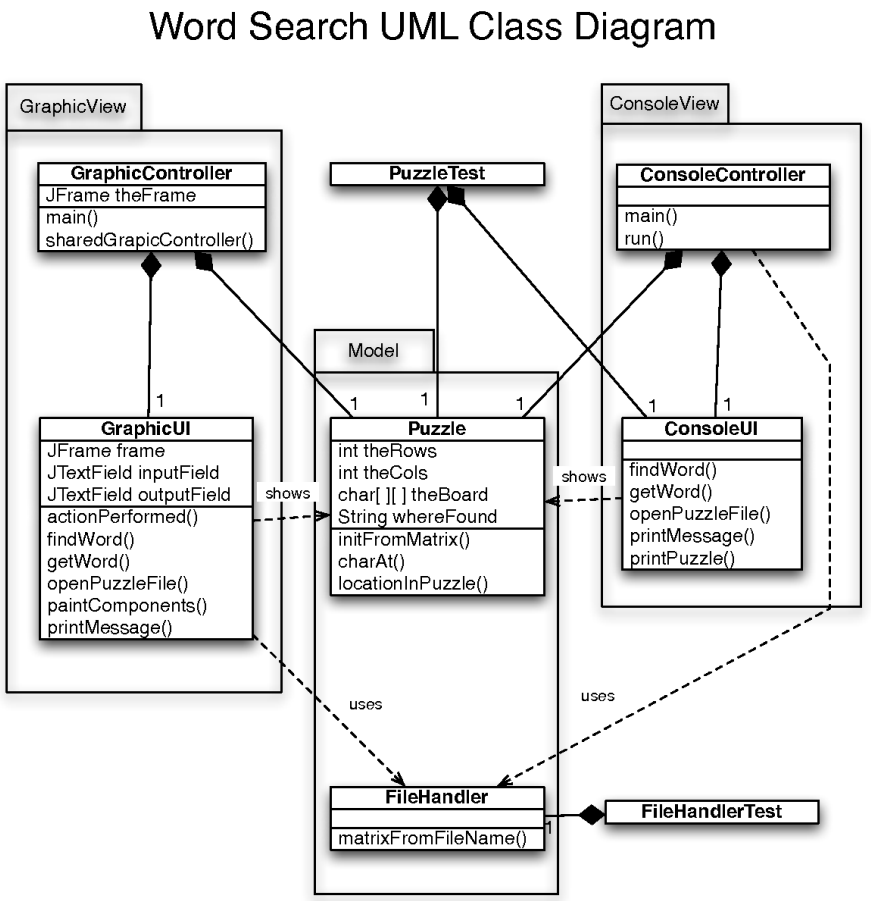
Example: UML class diagram



Example: UML class diagram



Example: UML class diagram (word search)



`std::string`

- C string is very... primitive, and there should be a better way to handle string
- `std::string` class provides convenient features, so that you don't have worry about detailed implementation (an example of abstraction)
- Templates are used to provide additional string types, e.g.,
  - `std::wstring`
  - `std::u16string`
  - `std::u32string`
- But using Korean characters (a.k.a., 한글) may be tricky
  - You many need to know character encodings (or locale) of your development and user environment
  - Does C++ support unicode? Yes and No

Example: std::string

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s1; // variable declaration
    // variable declaration with initialization
    // string support many initialization methods
    string s2 = "Hello, world!";
    string s3("Hello, world!");
    string s4 {"Hello, world!"};

    // string operations
    s1 = s2 + s3;
    cout << s1 + " " + s1[0] << s1[1] << s1.at(2) << endl;
}
```

Example: std::string (cont.)

```
 // string with cin
string name;
string age;
// what would happen if you type "John Doe"
cout << "Enter your full name: ";
cin >> name;
cout << "Enter your age: ";
cin >> age;
cout << name << " is " << age << " years old.\n";

// right way to handle multiple words into one string
// but be careful when you use this right after using cin
cout << "Enter your full name: ";
getline(cin, name); // read a full line
cout << name << endl;
```

Example: `std::string` (cont.)

```
// which one do you prefer, C-string or C++-string?
string arr[] =
    {"I ", "am ", "familliar ", "with ", "C/C++ ", "pointers!"};
int n_words = sizeof(arr) / sizeof(string);
string buffer {};

for (int i = 0; i < n_words; i++)
{
    cout << "length of \"" + arr[i] << "\": " << arr[i].length() << endl;
    buffer += arr[i];
}
cout << buffer << "\n";
```

Example: `std::string` (cont.)

```
// may work, or may not work, depending on your system
string hangul = "한글은 과학적 창제원리에 따라 만들어졌습니다.";
cout << hangul << endl;
```

Aliasing of types

- `typedef` (C/C++): put a new type name in the place of variable name in a declaration statement of the given type
- `using` (C++ only): put a new type name, following by = and old type
- Example

```
typedef long int lint;
using Lint = long int;
```

`static` member

- *A static member* is a variable that is a part of a class, but is not a part of an object of that class
- *A static member function* is a function that needs access to members of class, yet doesn't need to be invoked for a particular object

Example: `static` member

```
class Static {
public:
    static int s_val; // declaration
    int val;
    Static(int val) { this->val = val; s_val++; }
    static void set_sval(int s_val_) { s_val = s_val_; }
};
int Static::s_val; // definition
int main() {
    Static s1(42); Static s2(23);
    cout << s1.val << " " << s1.s_val << endl;
    cout << s2.val << " " << s2.s_val << endl;
    s1.set_sval(7);
    cout << Static::s_val << endl;
}
```

Reading list

- Learn C++
  - `this`, static member variables/functions: Ch. 8.8-12
  - template: Ch. 13
  - string: Ch. 4.4b
- Reference on character encodings
  - Templates: https://isocpp.org/wiki/faq/templates
  - Unicode in C++: https://youtu.be/MW884pluTw8
    - History and details of character encodings and C/C++ string
    - A little bit long (1:29:40, yes 1.5 hours, not 1.5 minutes)
  - 한글 인코딩의 이해
    - http://d2.naver.com/helloworld/19187
    - http://d2.naver.com/helloworld/76650

ANY QUESTIONS?