Object-Oriented Programming

UML Generalization

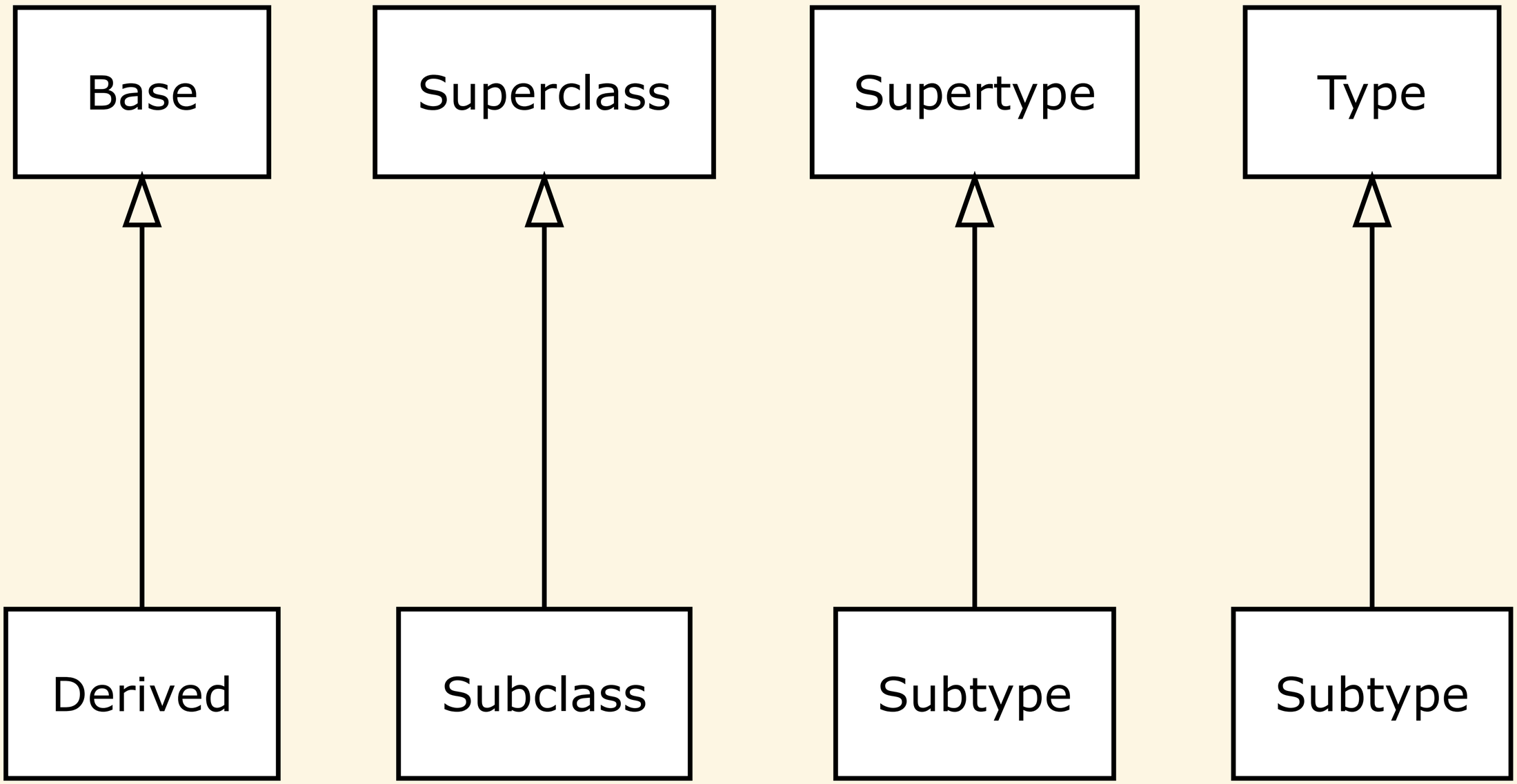
Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

UML Class Relationships

Relationship	**Description**
Dependency	*uses a*
Generalization	*is a*
Association	*has a*
Aggregation (Association)	*has a*
Composition (Association)	*has a*

Generalization Terminology

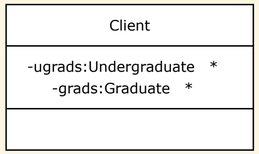


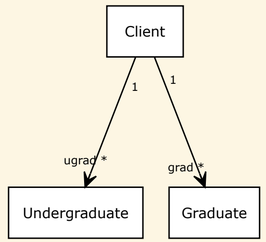
Problems with Generalization

The purpose of generalization is to solve a design problem. If you don't have a design problem, don't use generalization.

- What design problems does generalization solve?

Scenario: Multiple Related Classes





```
// get tuition from all students
for (const auto& student : grads) {
        // ...
}
for (const auto& student : ugrads) {
        // ...
}
```

Problems

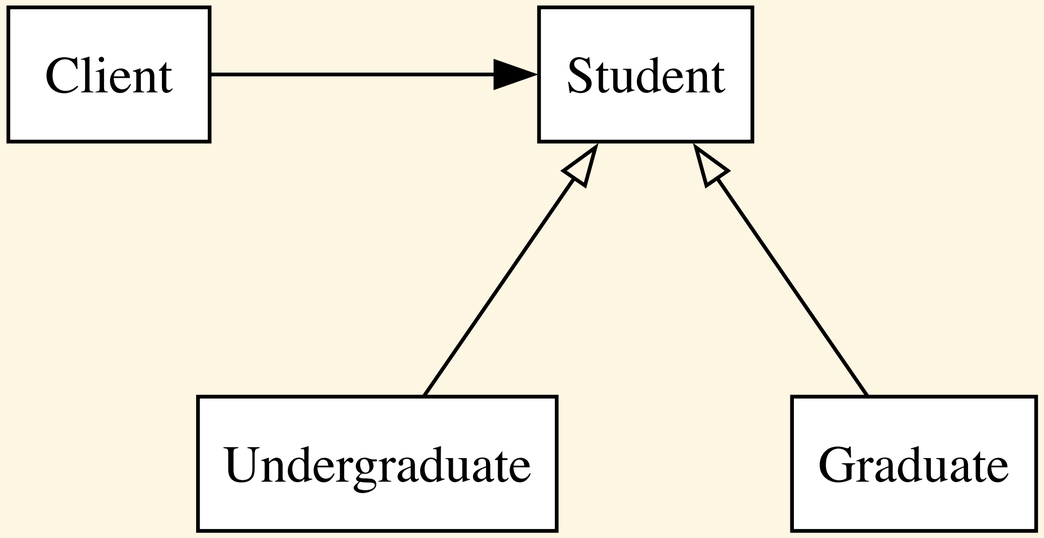
- If new kinds of students are added, we have to add more loops
- Every new processing we add has to know about both undergraduate and graduate students
- Have to remember to add new operations to both
- Three different concepts:

  Undergraduates

  Graduates

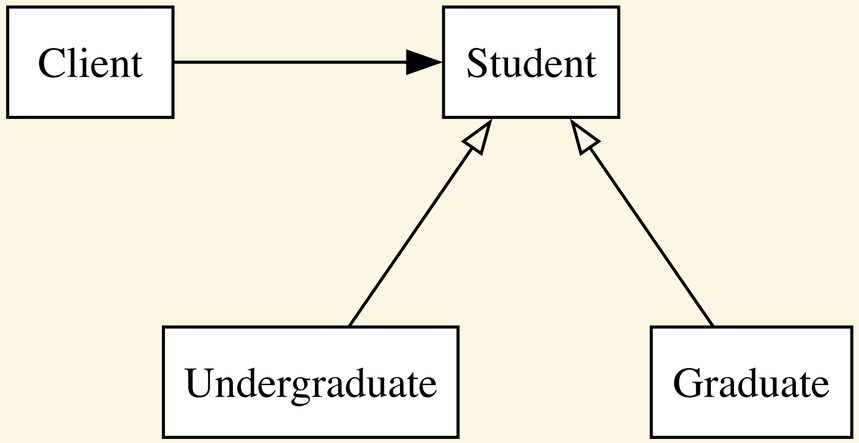
  Students - applies to both undergraduate and graduates

Solution




```
// get tuition from all students
for (const auto& student : students) {
        // ...
}
```

Solution



- Client *has a* Student (Association) but is totally decoupled from Undergraduate/Graduate

- Does Liskov hold? An *is a* relationship?

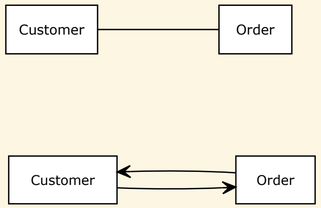
  ✓ *An undergraduate **is a** student*

  ✓ *A graduate student **is a** student*

- Single piece of code to manage any type of Student

- New kinds of Student can be added without changing the client code

Bidirectional Association



- Customer *has a*n Order

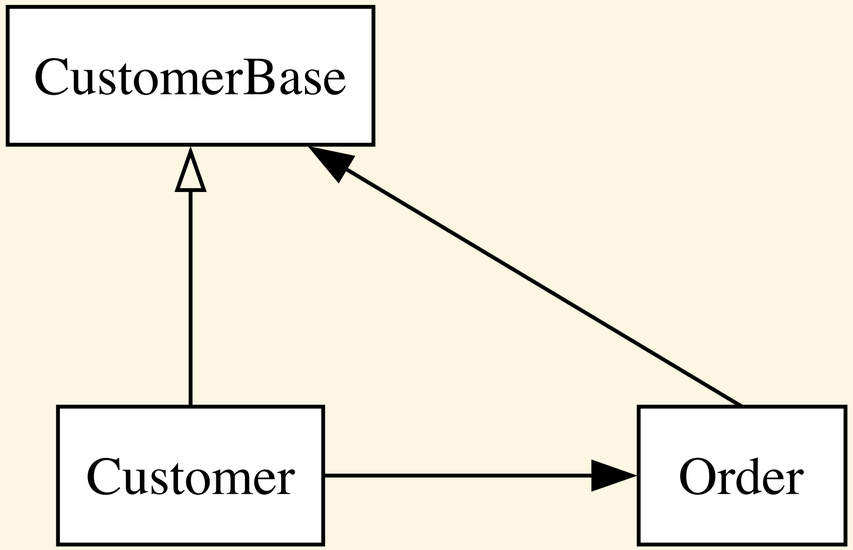
- Order *has a* Customer

Problems with Bidirectional Associations

```cpp
#ifndef INCLUDED_CUSTOMER_HPP
#define INCLUDED_CUSTOMER_HPP

#include <vector>
#include "Order.hpp"

class Customer {
private:
    std::vector<Order> Order;
};

#endif
```

```cpp
#ifndef INCLUDED_ORDER_HPP
#define INCLUDED_ORDER_HPP

#include "Customer.hpp"

class Order {
private:
    Customer id;
};

#endif
```

Solution



- Extract only the part needed
- Customer *has a*n order
- Order *has a* CustomerBase
- Customer *is a* CustomerBase

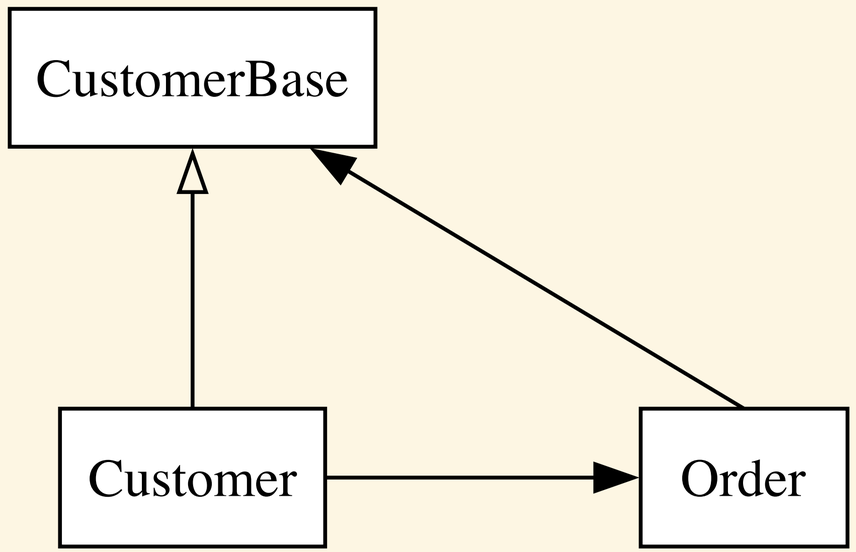
Solution Code

```hpp
#ifndef INCLUDED_CUSTOMERBASE_HPP
#define INCLUDED_CUSTOMERBASE_HPP

class CustomerBase {};

#endif
```

```hpp
#ifndef INCLUDED_CUSTOMER_HPP
#define INCLUDED_CUSTOMER_HPP

#include "CustomerBase.hpp"
#include "Order.hpp"
#include <vector>

class Customer : public CustomerBase {
private:
    std::vector<Order> Order;
};

#endif
```

```hpp
#ifndef INCLUDED_ORDER_HPP
#define INCLUDED_ORDER_HPP

class CustomerBase;

class Order {
private:
    CustomerBase& id;
};

#endif
```

Advantages



- Develop and test class *CustomerBase* individually
- Develop and test classes *Order* & *CustomerBase* together
- Develop and test classes *Customer* & *CustomerBase* together
- Often, class *CustomerBase* would be very small and simple, mostly to define an interface

Polymorphism

In general: The condition of occurring in several different forms

For design: A single interface to entities of different types

- Same *interface* for different types
- Which code is run depends on the type of the object

Static Polymorphism

- Determined at compile time
- *static dispatch*
- In C++:

overloading (function and operator)

templates

Without and With Method Overloading

```cpp
class Parser {
public:
    int importFd      (int fd);
    int importFilename(const char* filename);
    int importMemory  (char** buffer, size_t* size);
};
```

```cpp
class Parser {
public:
    int import(int fd);
    int import(const char* filename);
    int import(char** buffer, size_t* size);
};
```

Without and With Method Overloading

```cpp
Parser parser;

int output = STDOUT_FILENO;
parser.importFd(output);

const char* output = "overloading.cpp";
parser.importFilename(output);

char* buffer = nullptr;
size_t size = 0;
parser.importMemory(&buffer, &size);
```

```cpp
Parser parser;

int output = STDOUT_FILENO;
parser.import(output);

const char* output = "overloading.cpp";
parser.import(output);

char* buffer = nullptr;
size_t size = 0;
parser.import(&buffer, &size);
```

Method Overloading

```
Parser parser;

int output = STDOUT_FILENO;
parser.import(output);

const char* output = "overloading.cpp";
parser.import(output);

char* buffer = nullptr;
size_t size = 0;
parser.import(&buffer, &size);
```

- Code is more flexible as parameter types can change

- It is easier to remember the correct method names and parameters

- Completely determined at *compile-time*

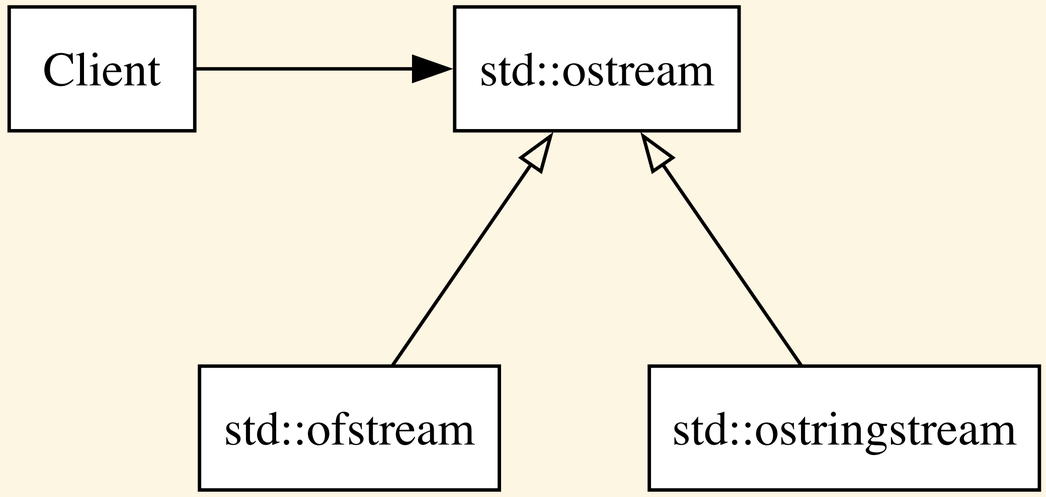
- Can disguise significant differences in behavior

- Can lead to bland and meaningless names, e.g., `process()`

Dynamic Polymorphism

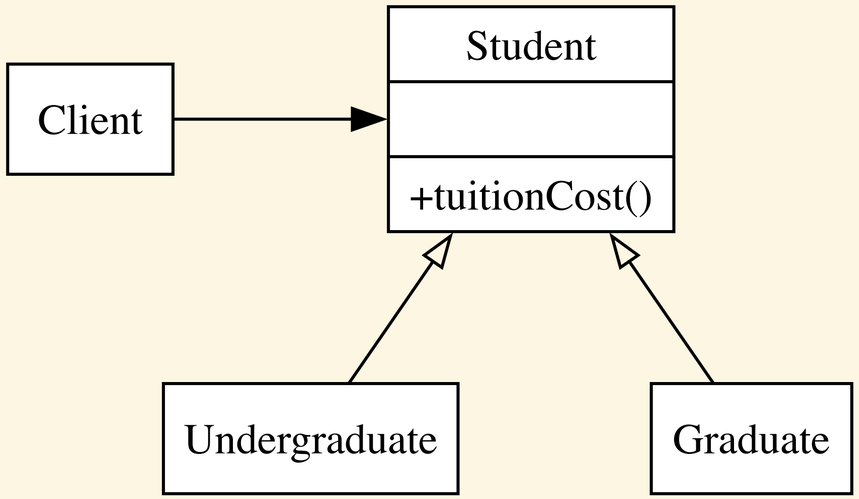
- Determined at run time
- *dynamic dispatch*
- In C++ requires `virtual` methods
- Lots of flexibility
- Slight speed disadvantage
- Extra memory to support the mechanism (vtables, vtable pointers)

Dynamic Polymorphism




```cpp
#include <iostream>
#include <fstream>
#include <sstream>

void output(std::ostream& out) {
    out << "Hello World!\n";
}

int main() {

    output(std::cout);

    std::ofstream out("outfile.txt");
    output(out);

    std::ostringstream outs;
    output(outs);

    return 0;
}
```

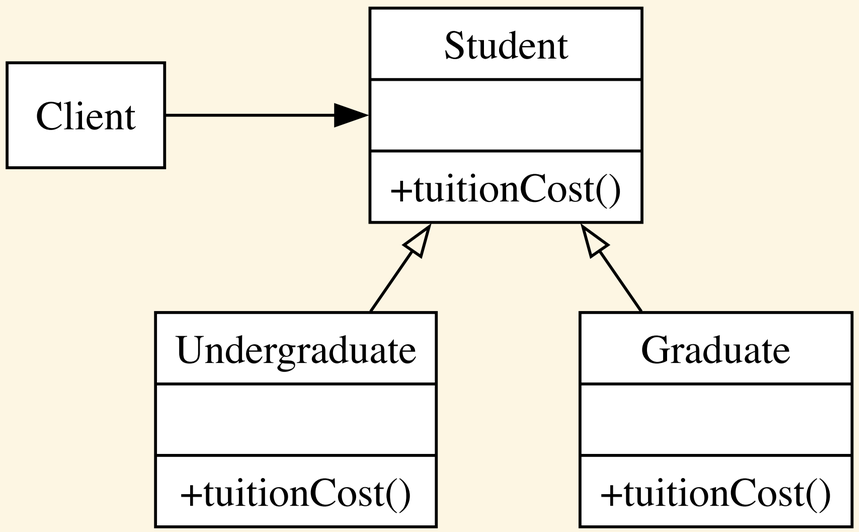
Back To Students: What about Methods?



- Of course, any Student object has `tuitionCost()`

- Due to the *is a* relationship, so does any Undergraduate or Graduate object

Scenario



- What if the classes Undergraduate and Graduate need different implementations for `tuitionCost()`?
- Could add the method to each (and implement it differently)

override

An override is when a derived class redefines a virtual method of the base class

- With UML Generalization, we assume that all methods are virtual

- Overriding is what provides the dynamic polymorphism

- Non-virtual and static methods **cannot** be overridden in C++

C++

```cpp
class Student {
public:
    virtual int costTuition() const;
};

class Undergraduate : public Student {
public:
    int costTuition() const;
};

class Graduate : public Student {
public:
    int costTuition() const;
};
```

- Once a method is virtual in a base class, it is virtual in all derived classes

- The `virtual` specifier is often also applied to the method in the derived class, but that is not necessary. A better approach is to use the `override` specifier placed after the `const`.

Design Questions

```
class Student {
public:
    virtual int costTuition() const;
};

class Undergraduate : public Student {
public:
    int costTuition() const override;
};

class Graduate : public Student {
public:
    int costTuition() const override;
};
```

- Does it make sense to have a student who is neither an undergraduate nor a graduate student?

- Perhaps all students should be one or the other? (at least for now)

- How do these get formed? How are the correct objects created?

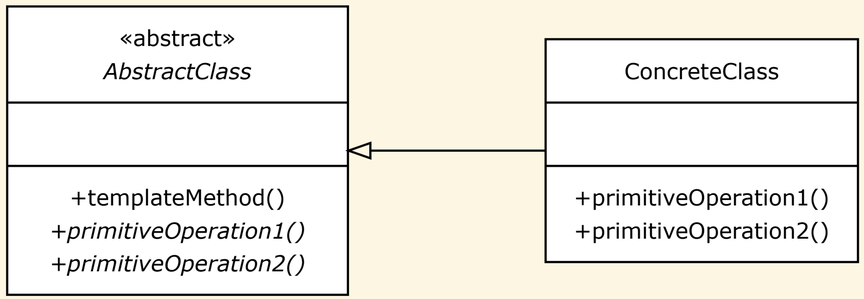
Methods

```cpp
// @NOTE Non-Virtual Method
class CustomerBase {
public:
    int getID() { return ID; }
private:
    int ID;
};
```

```cpp
// @NOTE Virtual Method
class CustomerBase {
public:
    virtual int getID() { return ID; }
private:
    int ID;
};
```

```cpp
// @NOTE Pure-Virtual Method
class CustomerBase {
public:
    virtual int getID() = 0;
};
```

Abstract Class

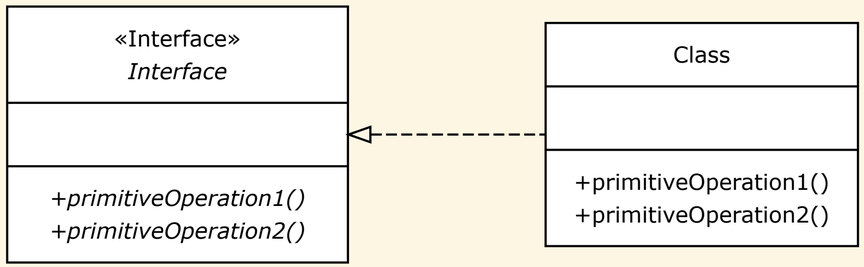


- Consists of at least one pure-virtual method

- Cannot create objects of that class type

- Can only create objects of derived classes that implement all the pure-virtual methods

Interface



- Consists of all pure-virtual methods
- Sometimes, all empty methods
- However, many times, the term is used interchangeably with *Abstract Class*

Virtual Design Choice

```cpp
// @NOTE Pure-Virtual Method
class CustomerBase {
public:
    virtual int getID() = 0;
};
```

```cpp
// @NOTE Empty Virtual Method
class CustomerBase {
public:
    virtual int getID() { return 0; }
};
```

Pure Virtual Design

```
// @NOTE Pure-Virtual Method
class CustomerBase {
public:
    virtual int getID() = 0;
};
```

- No pure `CustomerBase` objects

- A derived class *must* override and implement all pure virtual methods to create objects

- If all methods are *pure virtual*, then we potentially have an *interface*

Empty Virtual Design

```
// @NOTE Empty Virtual Method
class CustomerBase {
public:
    virtual int getID() { return 0; }
};
```

- Pure `CustomerBase` objects
- A derived class picks and chooses which methods to *override*