CS116

Public, Private& Protected

- public -> type inheritance
- private -> implementation inheritance
- protected -> ?

Inheritance Matrix

- **Individual Methods**

(inheritance method)			
private	private	private	X
protected	protected	protected	X
public	public	protected	X
	public	protected	private *(access method base class)*

- **Interface ?**

Object Composition

- Composition by value
  - other objects are declared members
- Composition by reference
  - only references or pointer are stored to other objects

Inheritance&Composition

```
class Stack {
  List l;
 public:
  Stack () { }
  Stack (Stack& q) throw (ListException) { l = (List&)q; }

  void Push (int) throw (ListException);
  void Pop (int&) throw (ListException);
  void Top (int&) throw (ListException);
  void Empty () { l.Empty (); }
  bool isEmpty () { return l.isEmpty (); }

  Stack& operator = (const Stack&) throw (ListException);
  friend std::ostream& operator << (std::ostream& out, Stack& q);
};
```

Inheritance&Composition

```
class Stack {
  List *l;
 public:
  Stack () { l = new List (); }
  Stack (Stack& q) throw (ListException) { l = new List ((List&)q); }

  void Push (int) throw (ListException);
  void Pop (int&) throw (ListException);
  void Top (int&) throw (ListException);
  void Empty () { l->Empty (); }
  bool isEmpty () { return l->isEmpty (); }

  Stack& operator = (const Stack&) throw (ListException);
  friend std::ostream& operator << (std::ostream& out, Stack& q);
};
```

Inheritance&Composition

```
class Stack : protected List {
 public:
  Stack () : List () { }
  Stack (Stack& q) throw (ListException) : List (q) { }

  void Push (int) throw (ListException);
  void Pop (int&) throw (ListException);
  void Top (int&) throw (ListException);
  void Empty () { List::Empty (); }
  bool isEmpty () { return List::isEmpty (); }

  Stack& operator = (const Stack&) throw (ListException);
  friend std::ostream& operator << (std::ostream& out, Stack&
q);
};
```

Virtual Functions

```
class A {
public:
   virtual void print () { cout << " this is an object of type A\n"; }

};

class B : A{
public:
   void print () { cout << " this is an object of type B\n"; }

};
```

Virtual Constructors (?)

```
class A {
public:
   virtual A () { cout << " create an object of type A\n"; }

};

class B : A{
public:
   B () : A () { cout << " create an object of type B\n"; }

};
```

Virtual Destructors

```
class A {
public:
  virtual ~ A () { cout << " destroy an object of type A\n"; }

};

class B : A{
public:
   ~B () { cout << " destroy an object of type B\n"; }

};

A *a = new B;
delete a;
```

Virtual Operators

```
class A {
public:
  virtual A& operator = (A&) { cout << " operator A&= called \n"; }
};

class B : public A {
public:
  A& operator = (A&) { cout << " operator BA&= called \n"; }
  B& operator = (B&) { cout << " operator B&= called \n"; }
};

B b1, b2;
A *a = &b1;

*a = b2;
```

dynamic_cast<> ()

- Convert a pointer to an object in a class hierarchy to a pointer of a different class in the same class hierarchy

- Convert a lvalue to a reference (objects)

- Note: the class must be polymorphic (contains a virtual member)

- Format:
- Class_name *ptr = dynamic_cast<Class_name*> (ptr_expr);
- Class_name &ptr = dynamic_cast<Class_name&> (ref_expr);

- Syntax:
  - A *a = new B();
  - B *b = dynamic_cast< B* > (a);

Virtual *new* Operator

```
class A {
public:
  virtual A* clone () = 0;
};

class B : A {
public:
   B () { cout << " create an object of type B\n"; }
   B (const A&) { cout << " create an object of type B and intialize\n"; }

   B *clone () { return new B (*this); }
};

B b; A *a = &b;
A *a2 = new A(a);    // incorrect
A *a2 = a->clone (); // correct
```

Multiple Inheritance

```
class A {
public:
  A () { cout << " create an object of type A\n"; }
  ~A () { cout << " destroy an object of type A\n"; }
};

class B {
public:
  B () { cout << " create an object of type B\n"; }
  ~B () { cout << " destroy an object of type B\n"; }
};

class C : public A, public B {
public:
  C () : A(), B() { cout << " create an object of type C\n"; }
  ~C () { cout << " destroy an object of type C\n"; }
};
```

Virtual Inheritance

```
class A : public O {
public:
  A () { cout << " create an object of type A\n"; }
  ~A () { cout << " destroy an object of type A\n"; }
};

class B : public O {
public:
  B () { cout << " create an object of type B\n"; }
  ~B () { cout << " destroy an object of type B\n"; }
};

class C : public A, public B {
public:
  C () : A(), B(), O () { cout << " create an object of type C\n"; }
  ~C () { cout << " destroy an object of type C\n"; }
};
```

Virtual Base Class Member

```
class A : virtual public O {
public:
  A () { cout << " create an object of type A\n"; }
  ~A () { cout << " destroy an object of type A\n"; }
};

class B : virtual public O {
public:
  B () { cout << " create an object of type B\n"; }
  ~B () { cout << " destroy an object of type B\n"; }
};

class C : public A, public B {
public:
  C () : A(), B() { cout << " create an object of type C\n"; }
  ~C () { cout << " destroy an object of type C\n"; }
};
```

Inheritance & Scopes

```
class A {
public:
 voidprint () { cout << " print from A" << endl; }
};
class B : public A { };
class C {
public:
 void print () { cout << " print from C" << endl; }
};

class D : public C, public B {
 void print () { print (); } // compilation error ???
 void print2 () { A::print (); } // or B::print ();
};
```

Constr./Destr. Order

```
class A : virtual public O {
public:
  A () { cout << " create an object of type A\n"; }
  ~A () { cout << " destroy an object of type A\n"; }
};

class B : virtual public O {
public:
  B () { cout << " create an object of type B\n"; }
  ~B () { cout << " destroy an object of type B\n"; }
};

class C : virtual public A, public B {
public:
  C () : A(), B() { cout << " create an object of type C\n"; }
  ~C () { cout << " destroy an object of type C\n"; }
};
```