Practice of Programming 3
Class III

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

类

- 设计原则
  - public继承是is-a关系
  - 避免遮蔽继承而来的名字
  - 不要重新定义继承而来的非虚函数
  - 通过复合来构建has-a关系
  - private继承
  - 谨慎使用多重继承
- 面向对象建模
  - 类图
  - 顺序图

public继承是 is-a 关系


```
class Person {...};
class Student: public Person {...};
```

- 子类对象可以当做父类对象，但是反之不成立

```
void eat(const Person& p);          // anyone can eat
void study(const Student& s);       // only students study
Person p;                           // p is a Person
Student s;                          // s is a Student
eat(p);                             // fine, p is a Person
eat(s);                             // fine, s is a Student,
                                    // and a Student is-a Person
study(s);                           // fine
study(p);                           // error! p isn't a Student
```

public继承是 is-a 关系

- 问题

```
class Bird {
public:
  virtual void fly();                    // birds can fly
  ...
};

class Penguin:public Bird {              // penguins are birds
  ...
};
```

public继承是 is-a 关系

- 方案一

```
class Bird {
  ...                          // no fly function is declared
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {
  ...                          // no fly function is declared
```

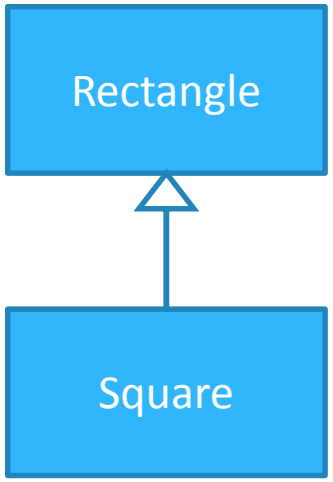
public继承是 is-a 关系

- 方案二

```
void error(const std::string& msg);      // defined elsewhere

class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin fly!");}
  ...
};
```

public继承是 is-a 关系

- 问题
  - Square是否应该继承自Rectangle？



public继承是 is-a 关系


```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);
  virtual int height() const;          // return current values
  virtual int width() const;
   ...
};

void makeBigger(Rectangle& r)          // function to increase r's area
{
   int oldHeight = r.height();
   r.setWidth(r.width() + 10);         // add 10 to r's width
   assert(r.height() == oldHeight);    // assert that r's
}
```

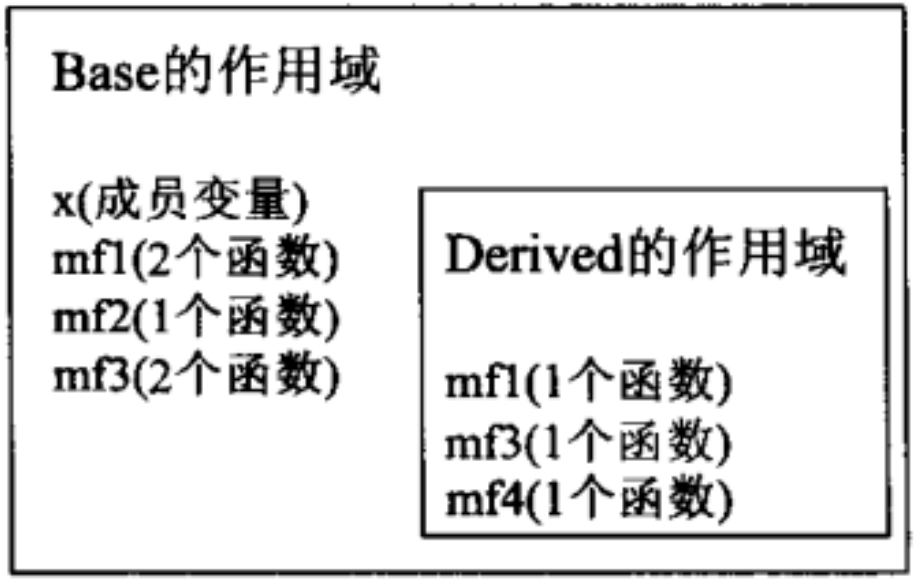
public继承是 is-a 关系


```
class Square: public Rectangle {...};

Square s;
...

assert(s.width() == s.height());  // this must be true for all squares
makeBigger(s);                    // by inheritance, s is-a Rectangle,
                                  // so we can increase its area
                                  // for all squares
```

- 请记住：public继承是is-a关系，每一个导出类对象都是一个基类对象，适用于基类的所有事情都需要适用于导出类。

避免遮蔽继承而来的名字


```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
};
```



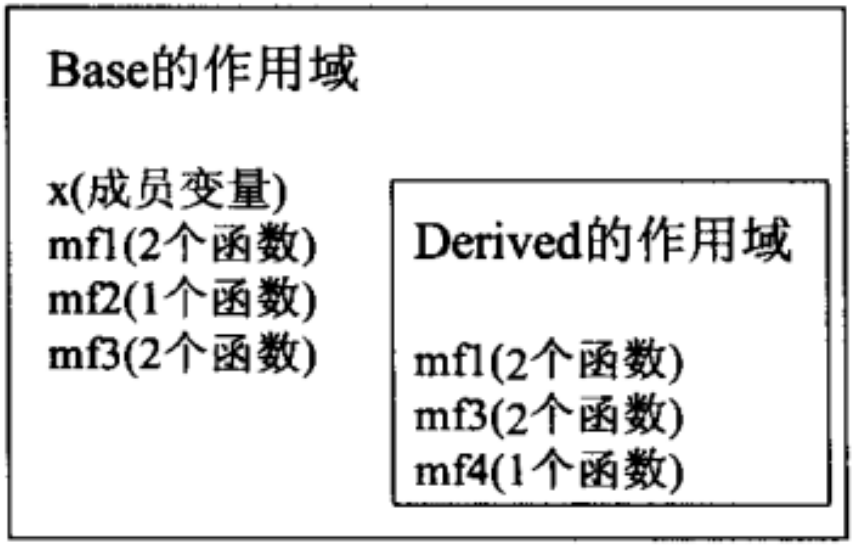
避免遮蔽继承而来的名字


```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
};
```

```
Derived d;
int x;
...

d.mf1();   // fine, calls Derived::mf1

d.mf1(x); // error! Derived::mf1 hides Base::mf1

d.mf2();   // fine, calls Base::mf2

d.mf3();  // fine, calls Derived::mf3

d.mf3(x);  // error! Derived::mf3 hides Base::mf3
```

避免遮蔽继承而来的名字


```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  using Base::mf1;      // make all things in Base named mf1 and mf3
  using Base::mf3;      // visible (and public) in Derived's scope
  virtual void mf1();
  void mf3();
  void mf4();
};
```



避免遮蔽继承而来的名字


```
class Base {
private:
  int x;
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};

class Derived: public Base {
public:
  using Base::mf1;
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4();
};
```

```
Derived d;
int x;
...

d.mf1();   // still fine, still calls Derived::mf1

d.mf1(x); // now okay, calls Base::mf1

d.mf2();   // still fine, still calls Base::mf2

d.mf3();  // fine, calls Derived::mf3

d.mf3(x);  // now okay, calls Base::mf3
```

避免遮蔽继承而来的名字


```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  ...                          // as before

};

class Derived: private Base {
public:
  virtual void mf1()           // forwarding function;
  { Base::mf1(); }
   ...
};

Derived d;
int x;
d.mf1();                       // fine, calls Derived::mf1
d.mf1(x);                      // error! Base::mf1() is hidden
```

避免遮蔽继承而来的名字

- 请记住
  - 导出类中的名字会遮蔽基类中的名字，不论是虚函数还是非虚函数
  - 可以使用using指示符或者使用forwarding function来解决问题

不要重新定义继承而来的非虚函数


```
class B {
public:
  void mf();
  ...
};

class D: public B {
public:
  void mf();                    // hides B::mf;
  ...
};

D x;                            // x is an object of type D
B *pB = &x;                     // get pointer to x
pB->mf();                       // call mf through pointer
D *pD = &x;                     // get pointer to x
pD->mf();                       // call mf through pointer
pB->mf();                       // calls B::mf
pD->mf();                       // calls D::mf
```

通过复合来构建has-a关系


```
class Address { ... };          // where someone lives
class PhoneNumber { ... };

class Person {
public:
  ...
private:
  std::string name;             // composed object
  Address address;              // ditto
  PhoneNumber voiceNumber;      // ditto
  PhoneNumber faxNumber;        // ditto
};
```

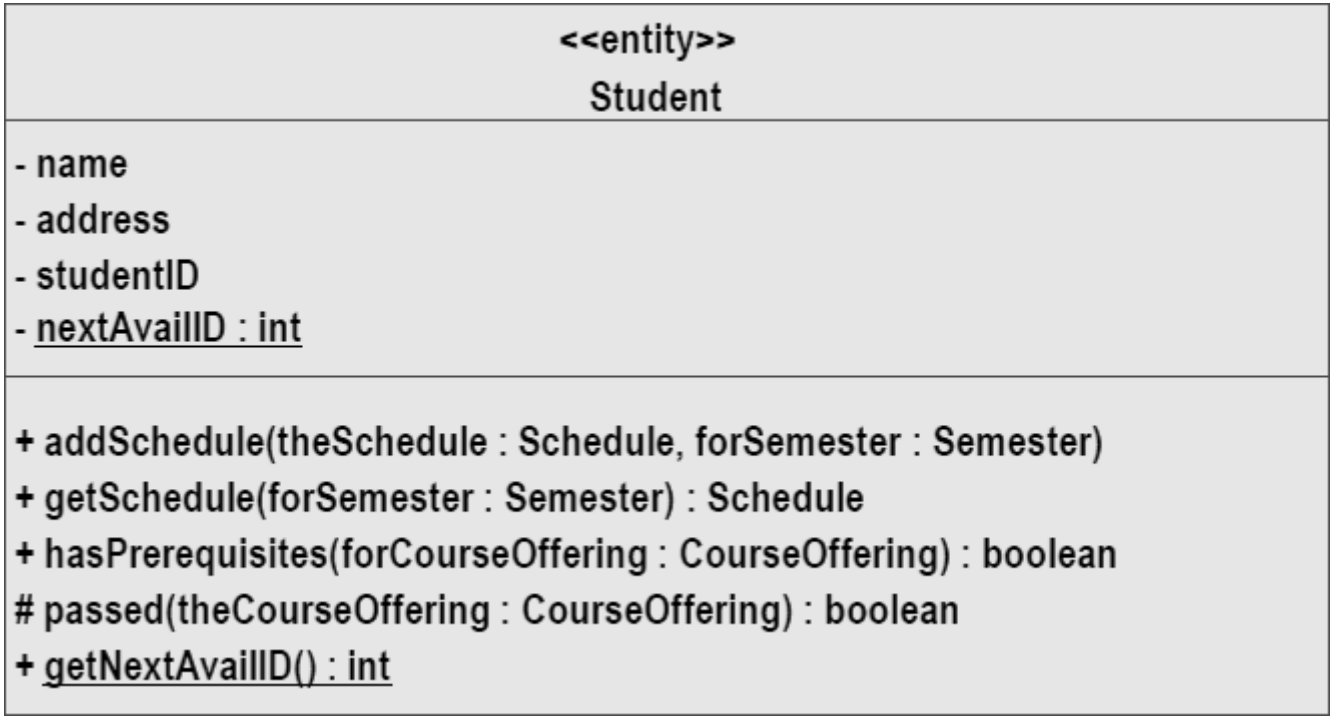
private继承


```
class Person { ... };
class Student: private Person { ... };    // inheritance is now private

void eat(const Person& p);                // anyone can eat
void study(const Student& s);             // only students study

Person p;                                 // p is a Person
Student s;                                // s is a Student

eat(p);                                   // fine, p is a Person
eat(s);                                   // error! a Student isn't a Person
```

谨慎使用多重继承

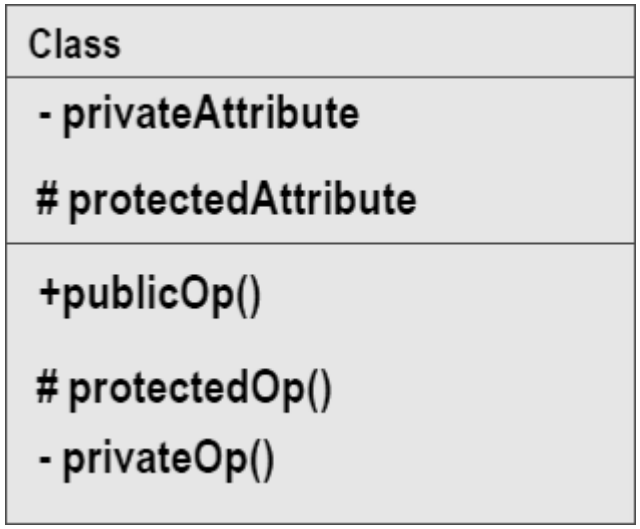

```
class BorrowableItem {          // something a library lets you borrow
public:
  void checkOut();              // check the item out from the library
  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;        // perform self-test, return whether
  ...                           // test succeeds
};

class MP3Player:                // note MI here
  public BorrowableItem,        // (some libraries loan MP3 players)
  public ElectronicGadget
{ ... };                        // class definition is unimportant

MP3Player mp;
mp.checkOut();                  // ambiguous! which checkOut?
```

UML Class Diagram



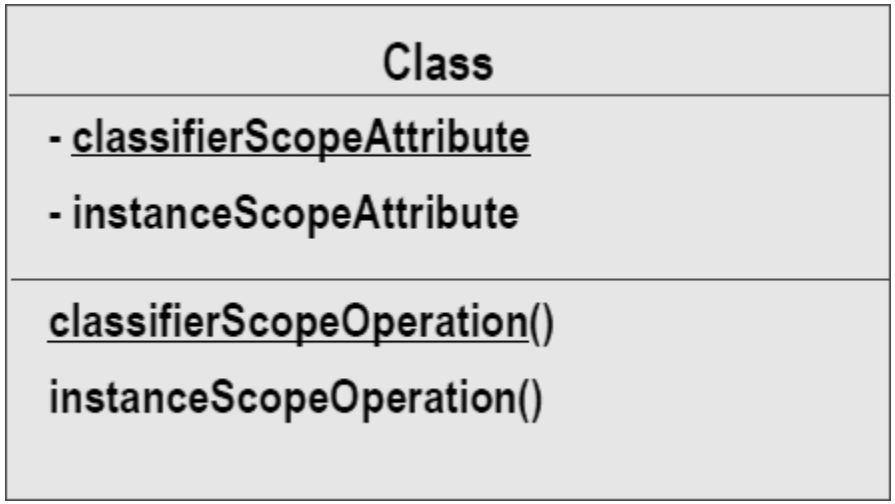
UML Class Diagram: How Is Visibility Noted?

- The following symbols are used to specify export control:
  - **+ Public access**
  - **# Protected access**
  - **- Private access**



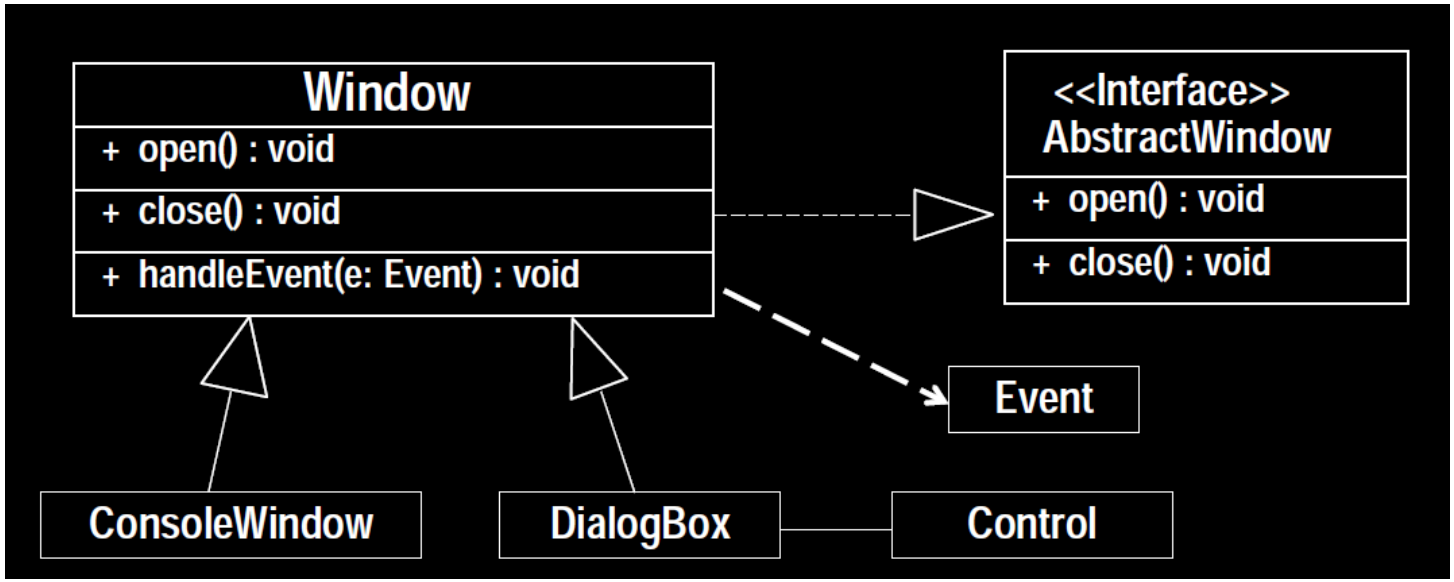
UML Class Diagram: Scope

- Determines number of instances of the attribute/operation
  - **Instance: one instance for each class instance**
  - **Classifier: one instance for all class instances**
- Classifier scope is denoted by underlining the attribute/operation name



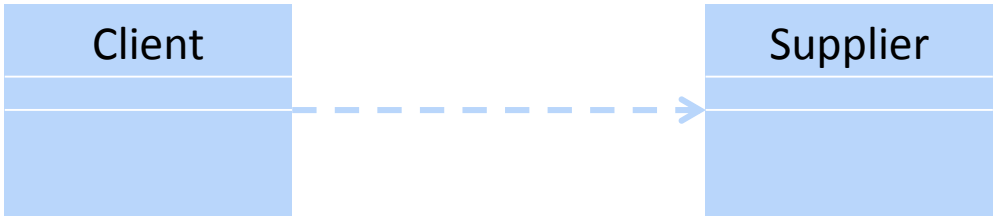
UML Class Diagram: Relationships

- There are four kinds of relationships in the UML:
  - Association
  - Dependency
  - Generalization
  - Realization



UML Class Diagram: Define Dependency

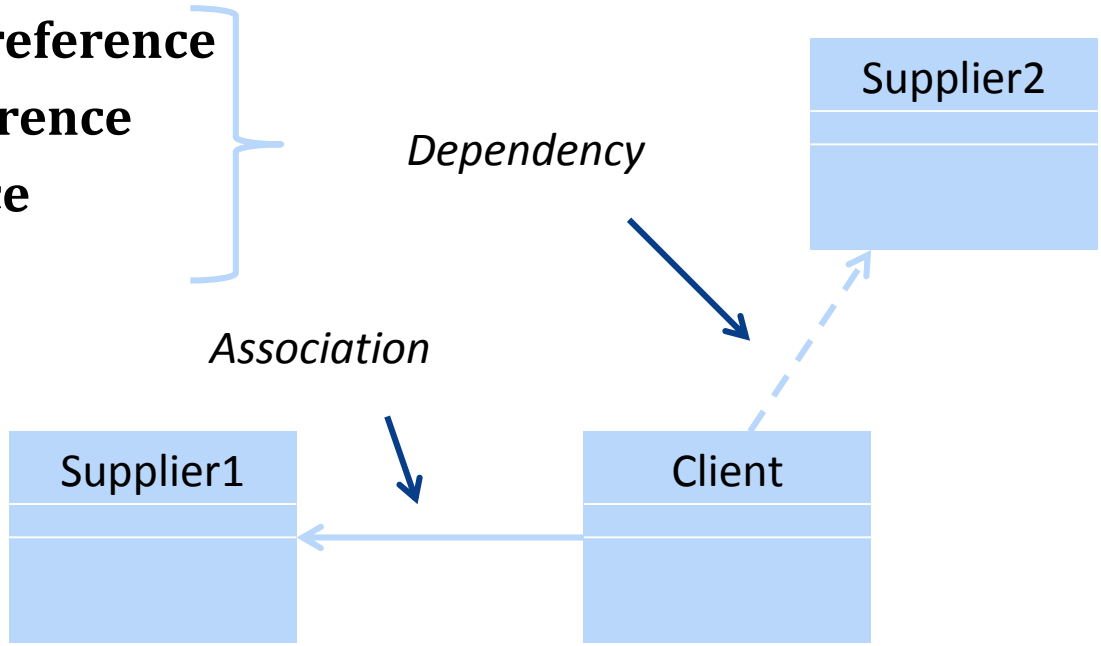
- What Is a Dependency?
  - **A relationship between two objects**



- Purpose
  - **Determine where structural relationships are NOT required**
- Things to look for :
  - **What causes the supplier to be visible to the client**

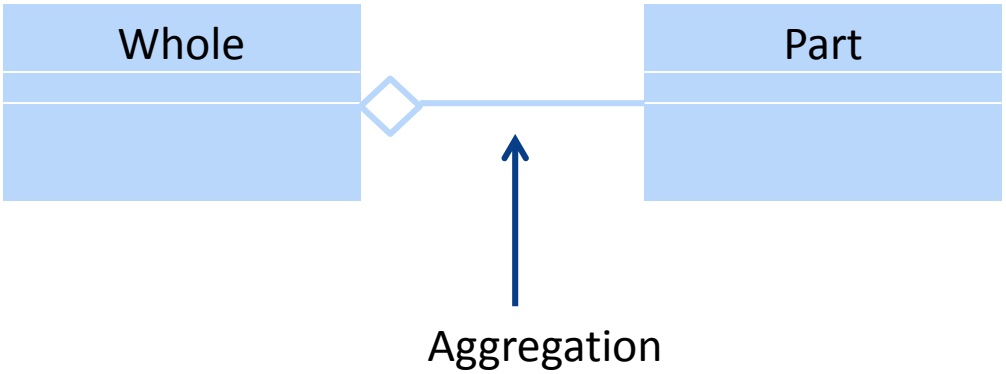
UML Class Diagram: Dependencies vs. Associations

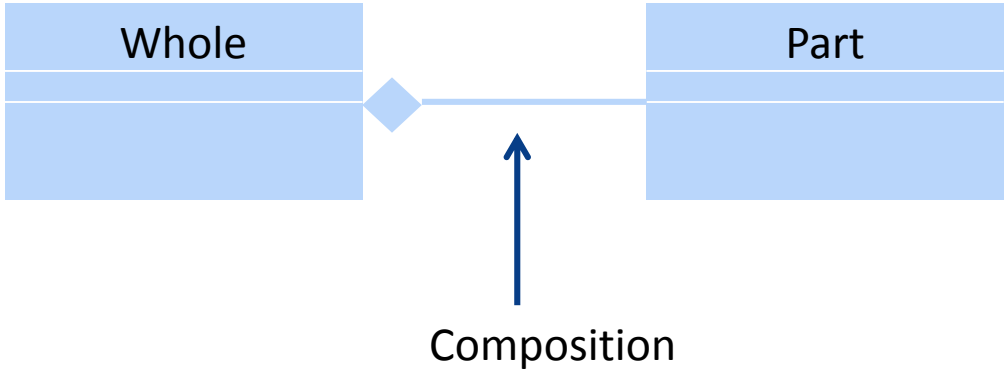
- Associations are structural relationships
- Dependencies are non-structural relationships
- In order for objects to "know each other" they must be visible
  - **Local variable reference**
  - **Parameter reference**
  - **Global reference**
  - **Field reference**



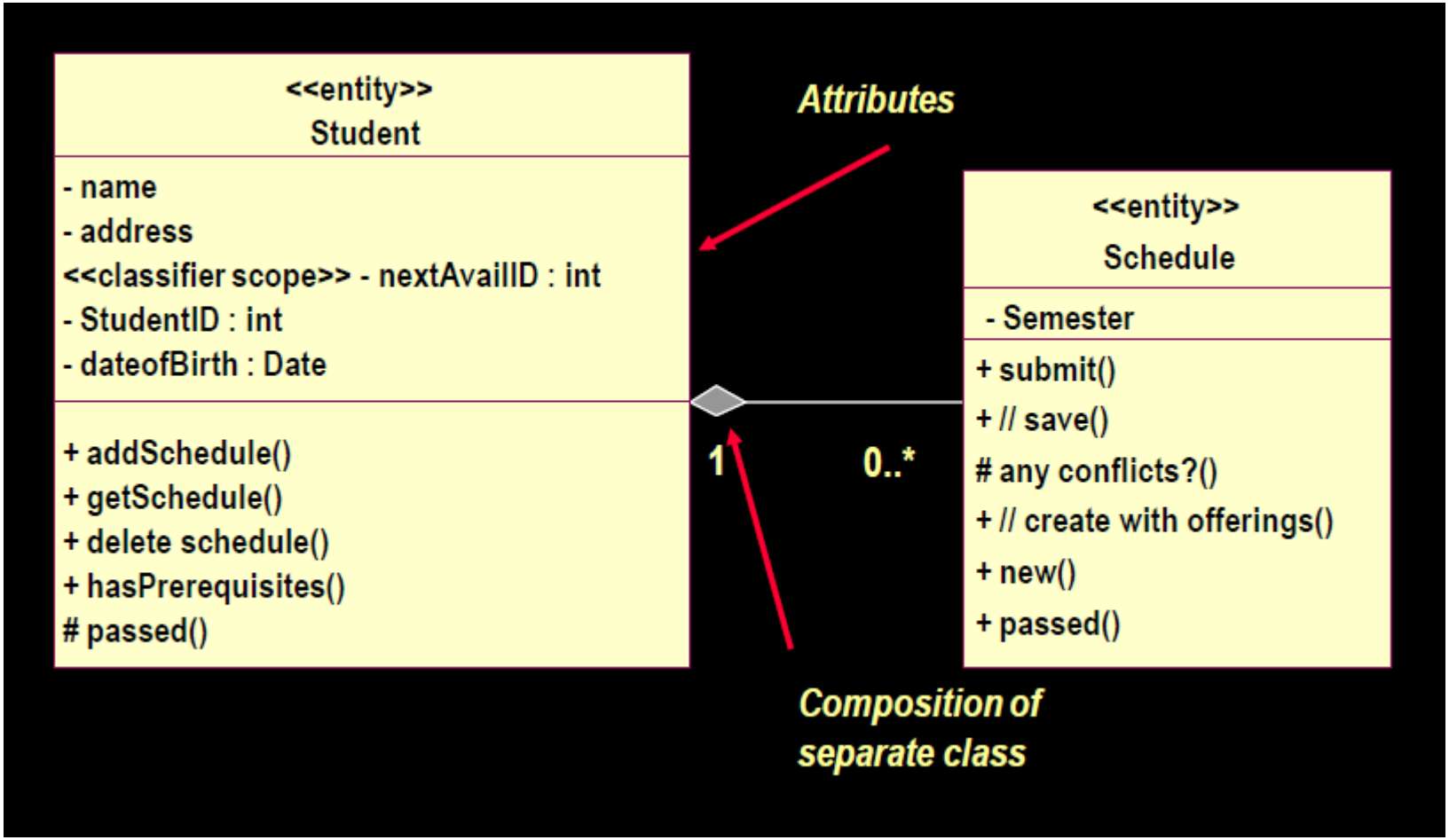
UML Class Diagram: Aggregation or Composition?

- Consideration
  - **Lifetimes of Class1 and Class2**



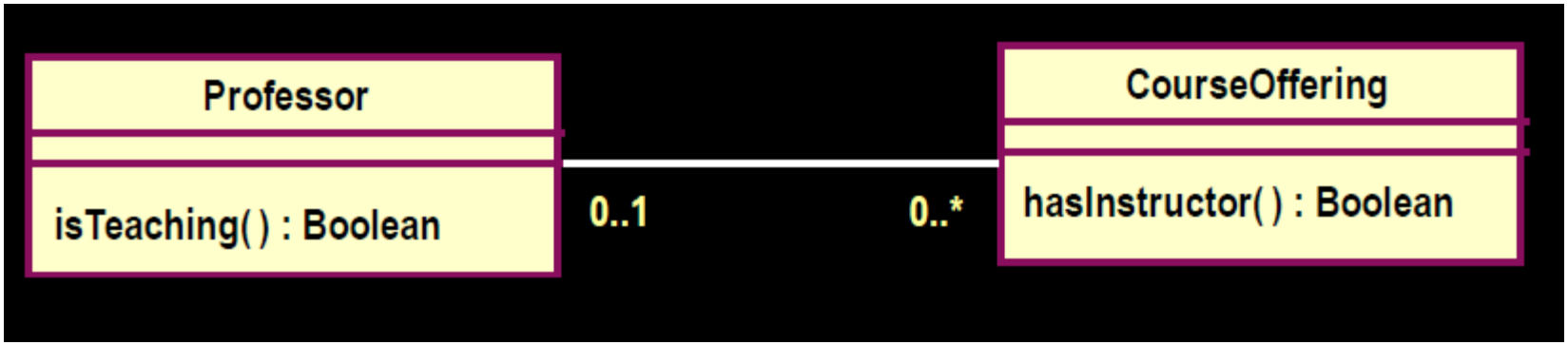


UML Class Diagram: Aggregation or Composition?



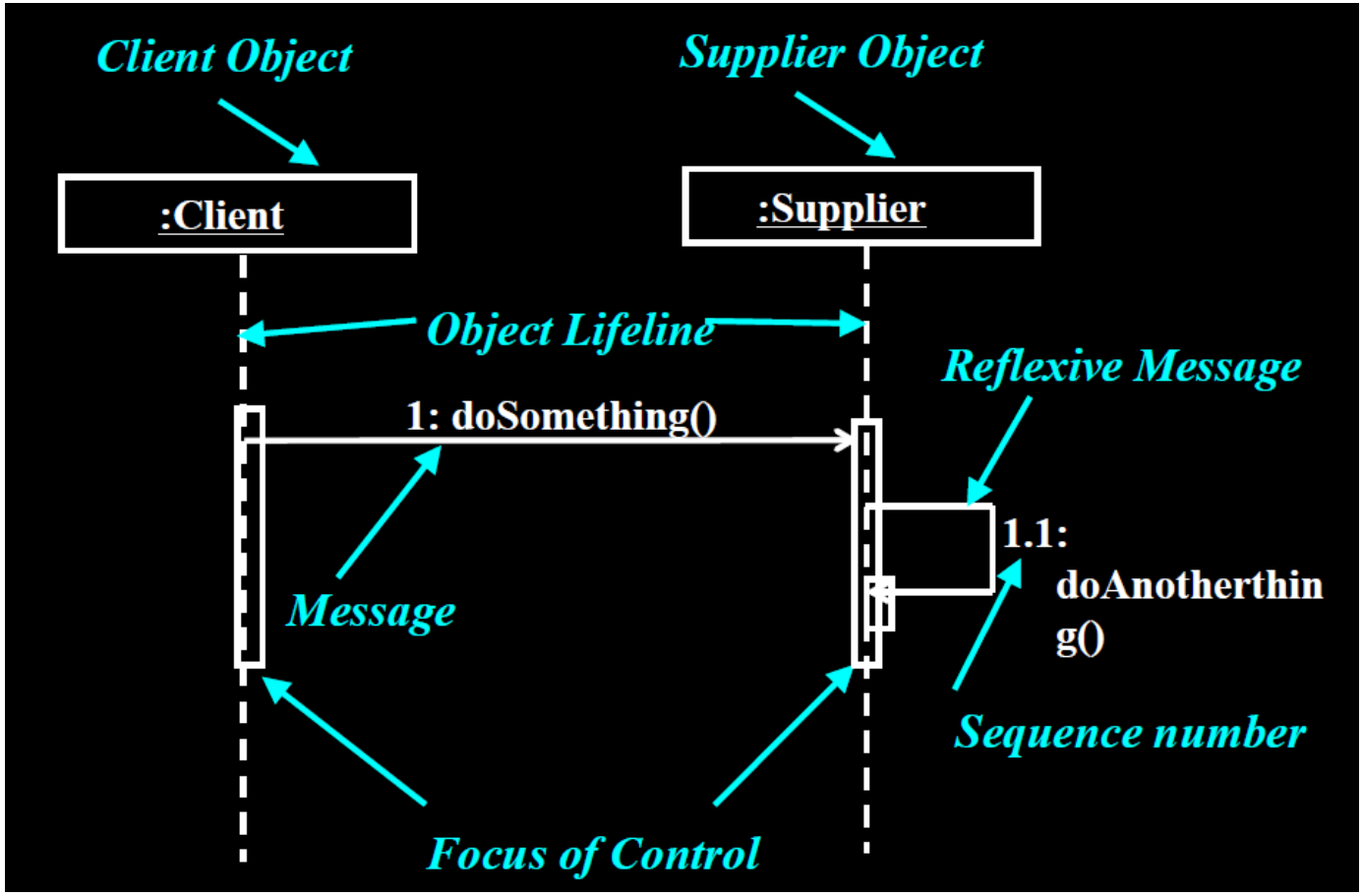
UML Class Diagram: Multiplicity Design: Optionality

- If a link is optional, make sure to include an operation to test for the existence of the link



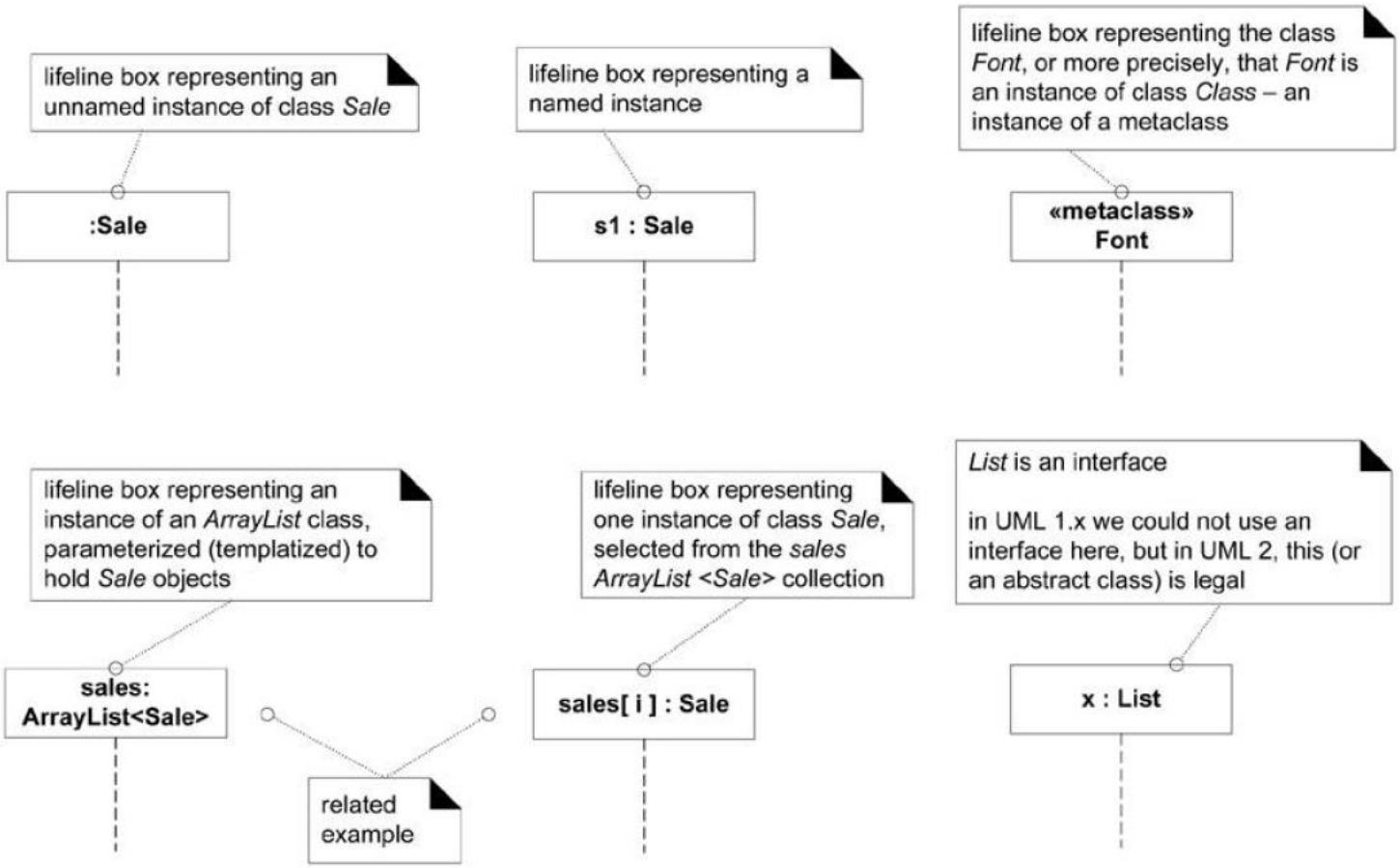
UML Sequence Diagram

- A sequence diagram emphasizes the time ordering of messages.

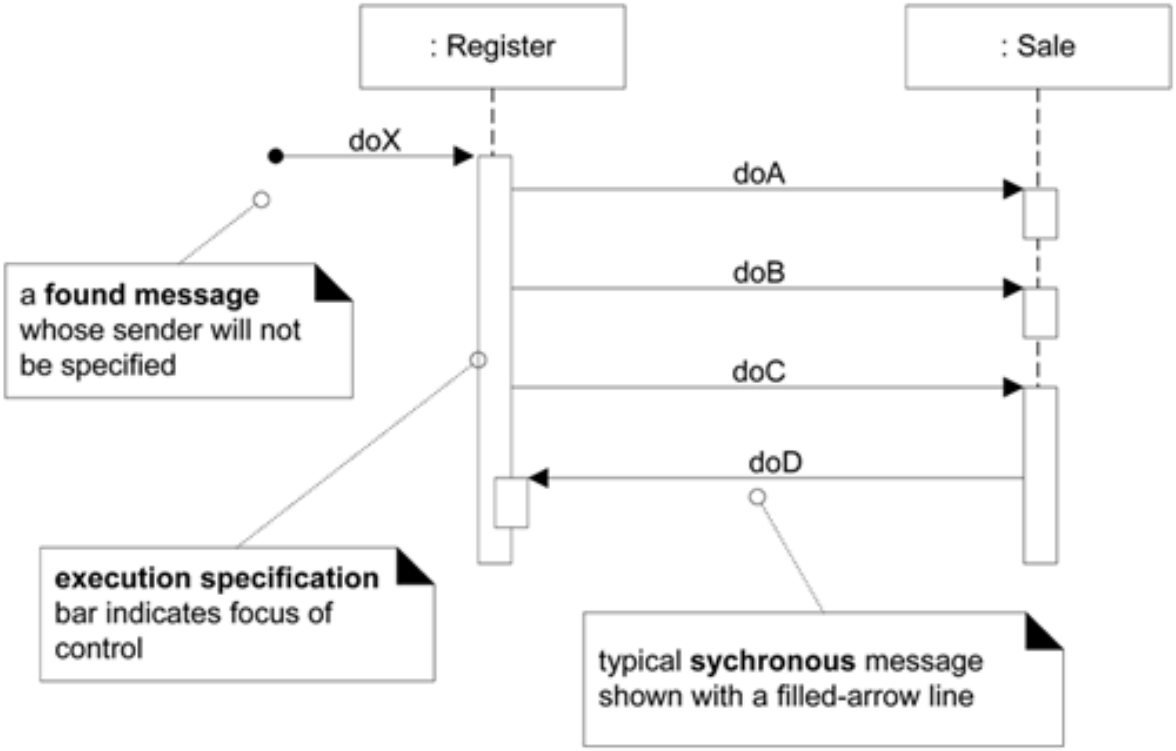


UML Sequence Diagram: lifeline

- A lifeline represents an individual participant in the Interaction.

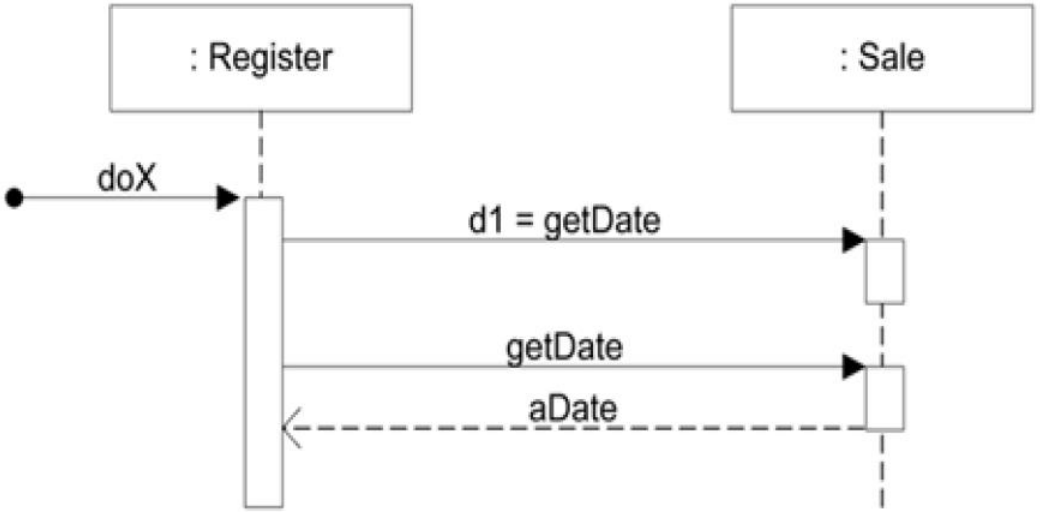


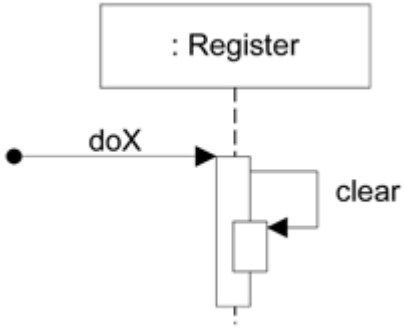
UML Sequence Diagram: Messages



UML Sequence Diagram: Messages

- Two ways to show a return result from a message
  - Using the message syntax returnVar = message(parameter).
  - Using a reply (or return) message line

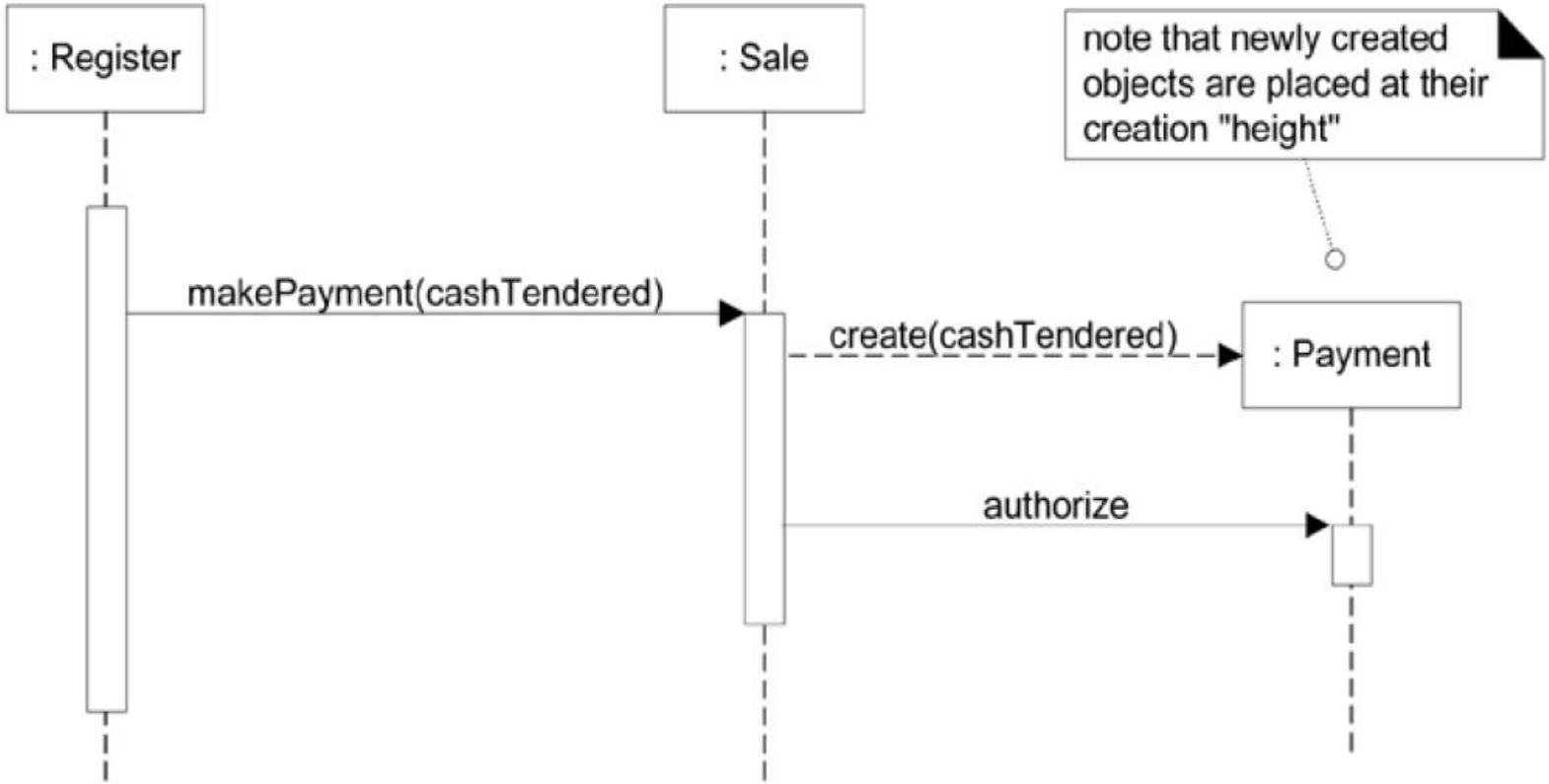




- Messages to "self" or "this"
  - You can show a message being sent from an object to itself by using a nested activation bar

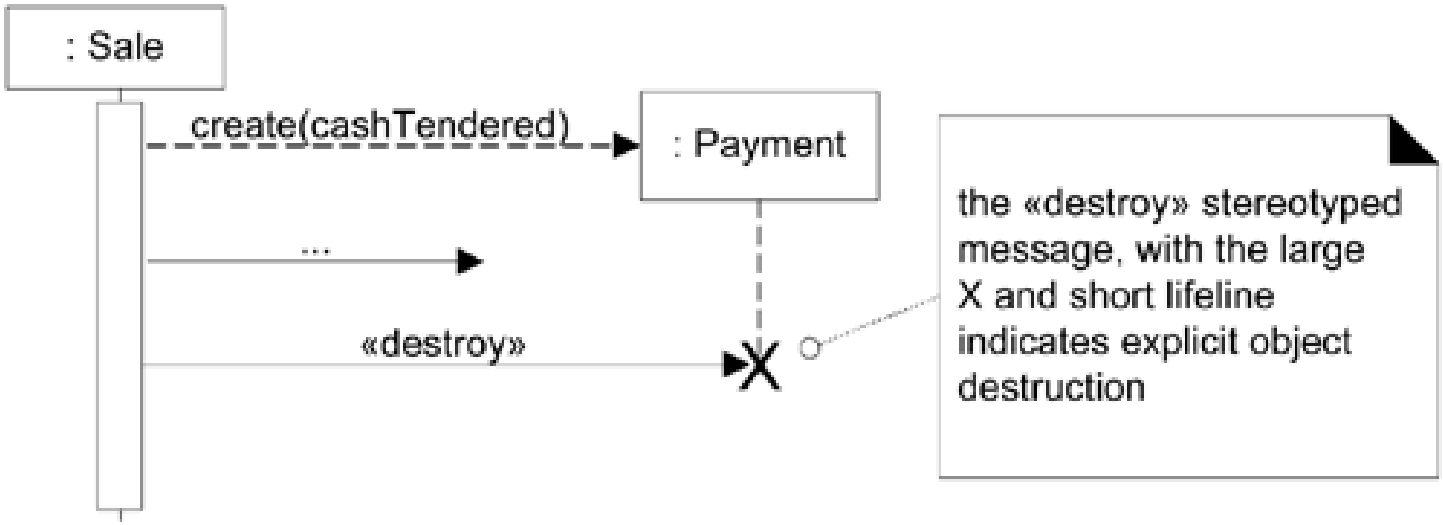
UML Sequence Diagram: Messages

- Creation of Instances



UML Sequence Diagram: Messages

- Object Destruction



Thank You!