C++ & OOAD

Ajay

Introduction to C++

- Understanding Encapsulation with Classes and Objects
- How objects use functions
- Dynamic Memory & References
- Initialization with constructors
  - Types of constructors
- Clean up using destructors

Characteristics of OOP

All object-oriented programming languages have three traits in common:

1. Encapsulation.

2. Polymorphism.

3. Inheritance.

Introduction Object Oriented Programming

The fundamental idea behind **OOP** is to combine into a single unit

Data

And

Functions that operate on that data

Classes & Objects

- A class has 3 parts:
  - ① A Name
  - ② Data Members
  - ③ Functions

Class declaration

```
#ifndef MY_HEADER
#define MY_HEADER
class IdGenerator {
public:
        // public members declaration
protected:
        // protected members declaration
private:
        // private members declaration
};

#endif
```

Let’s declare a class

```
#define SIZE 100
class Stack {                              // encapsulates data &
   operations
private:
      int stck[SIZE];
      int tos;
public:
      void init();
      void push(int i);
      int pop();
};
```

Defining functions: outside / different file

```
void Stack :: init()                    // init() belongs to the class Stack
 {
        tos = 0;
}
void Stack :: push(int i)  {
      if(tos == SIZE) {
        cout << "stack is full" << endl;
      }
    else {
      stck[tos] = i;
      tos++;
    }
}
```

pop here..

```
int Stack :: pop()
{
    if(tos == 0){
      cout << "stack is empty" << endl;
      return -1;
    }
    tos--;
    return stck[tos];
}
```

Objects

- Objects are created from classes
  - An Object is an **instance** of a Class
  - Creating an Object from a Class is called instantiating
  - An object is automatically given all the **capabilities** and **characteristics** of its class
  - We use **dot** notation to refer to the object name and its properties / methods

Think of *class* as a new *datatype* and *object* as a *variable* of that datatype

Object = attributes + operations

```
Stack stack1;                // stack1 is an object of class Stack

stack1.init(); // invoke the member function using object dot function
```

```
    stack1.push(11);
    stack1.push(21);
    stack1.push(31);
    stack1.push(41);
    stack1.push(51);
    stack1.push(61);

    cout << stack1.pop() << " ";
    cout << stack1.pop() << " ";
    cout << stack1.pop() << " ";
    cout << stack1.pop() << " ";
```

```
Stack * sptr;
sptr = &stack1;
sptr -> push(20);
```

Think of class as a new datatype and object as a variable of that datatype

Reference = alias to an object

-Alias name given to an existing object
-Must be initialized at the time of creation

-**Use**

-Cleaner way to pass arguments to function

Syntax:

```
Datatype var1 = value;

Datatype  & ref1 = var1;     // ref1 is a reference variable to object var1
```

Using references

```
void negate_num( int & );

int main()
{
            int x;
            x = 10;

            cout << "x = " << x << endl;

            negate_num(x);          // call is easy

            cout <<  "negated x = " <<  x <<  endl;
            return 0;
}

void negate_num(int & i) // i is a reference to the argument passed
{
  i = -(i);
}
```

Return by reference

```
int & say_hello(int & x)
{
   x = x * x;
   return (x);

}

int main()
{
   int x;

   cin >> x;

   x = say_hello(x);

   cout << x << endl;

   return 0;
}
```

Caution : **reference to a local variable should not be returned**

new & delete for DMA

-**Single Object**

-Allocation

```
Datatype * ptr = new Datatype;
```

-Release

```
delete ptr;
```

-**Array of Objects**

-Allocation

```
Datatype  *ptr = new Datatype [ no_of_elements];
```

-Release

```
delete [ ] ptr;            // prevent memory leak
```

new and delete- better than malloc & free

-No need of typecasting with new

-No need of sizeof(), automatic size computation

-new Invokes constructor, malloc() does not

-delete invokes destructor, free() does not

-new can **initialize** also (single element only)

    -E.g.   int *iptr = new int (100);

Not to be confused with *new int [100]*

Initialization needed!!!

```
void Stack :: init()                        // init() belongs to the class Stack
 {
    tos = 0;
}

---------------

Stack stack1;                    // stack1 is an object of class Stack

stack1.init();  // what happens if I forget to invoke init() for stack1 object?
stack1. push(190);
stack1.pop();
```

Use Constructors for initialization.

A Constructor is a special member function whose task is to initialize the objects of its class.

It has the same name as that of class.

It is invoked automatically when the object is created.

Example for Stack class

```
#define SIZE 100
class Stack {                          // encapsulates data & operations
private:
        int stck[SIZE];
        int tos;
public:
        //void init();     // why to call separately?
        Stack();           // constructor of the class Stack
        void push(int i);
        int pop();
};
```

constructor

```
/*******

void Stack :: init()
{
     tos = 0;
}

********/

Stack :: Stack()        // Stack () belongs to class Stack
{
     tos = 0;           // do the initialization of the object's data.
}
```

noteworthy

- It has no explicit return type, not even void
- They are not inherited
- There can be multiple constructors in a class
- Constructors can have default arguments
- Their address cannot be used

1. **Constructor with no arguments(default constructor).**

called when an object is created with no specification

2. **Constructor with arguments(parameterized constructors)**

able to initialize our objects at definition time and specify values for respective data members.

3. **Copy constructor.**

-used to declare and initialize an object from another.

-It takes a reference to an object as an argument

```
class myRTOS  {
 private:
  int       reg_no;
  char  *ptr_name;
 public:
  myRTOS()      // default constructor
  {
           reg_no = 0;
           ptr_name = new char[20];
           strcpy(ptr_name, "test");
  }

  myRTOS(char *s, int num)   // parameterized
  {
           reg_no = num;
           ptr_name = new char[strlen(s)+1];
           strcpy(ptr_name,s);
  }
  void change_data() {
          strcpy(ptr_name, “amit”);
   }
```

Shallow copy

```
myRTOS obj1;

myRTOS obj2(“Ajaykumar”,2);

myRTOS obj3 = obj2;

obj2.change_data();
```

copy constructor to achieve deep copy

```
class myRTOS  {
  private:
    int       reg_no;
    char  *ptr_name;

  public:
    myRTOS() ;     // default constructor
    myRTOS(char *s, int num);  // parameterized
    myRTOS(const myRTOS &ob); // copy con
};

// copy constructor goes like this
myRTOS :: myRTOS( const  myRTOS  &ob )
{
                this->reg_no = ob.reg_no;
// now do the necessary allocation first
                ptr_name =
                 new char[strlen(ob.ptr_name)+1];
                strcpy(ptr_name, ob.ptr_name);
}
```

```
myRTOS obj1;

myRTOS obj2(“Ajaykumar”,2);

myRTOS obj3 = obj2;

// myRTOS obj3 (obj2);

obj2.change_data();
```

Copy constructor is invoked ….

- When an object is created and initialized with an existing object
- When an object is passed by value as a parameter to a function
- When an object is returned from a function
- When an object is inserted into a STL container

```
myRTOS  rob1;
vector<myRTOS> vob1;
vob1.push_back(rob1);     // invokes copy constructor of myRTOS
```

- When a STL container is declared with more than one object

```
vector<myRTOS>  vob(5);  // invokes copy constructor
```

-The complement of a constructor is a destructor.

-A destructor has the same name as the class preceded by a ~

-In many circumstances, an object needs to perform some action when it is destroyed (technically, when the object goes out of scope)

- When an object is destroyed, its destructor function (if one exists) is automatically called.

-Destructor function do not take any arguments

-do not have any return values.

Destructor – when & what?

```
class myRTOS  {
  private:
    int       reg_no;
    char  *ptr_name;

  public:
    myRTOS() ;     // default constructor
    myRTOS(char *s, int num);  // parameterized
    myRTOS(const myRTOS &ob); // copy con
};


    …………………………..
```

```
{
       myRTOS obj2(“Ajaykumar”,2);

       myRTOS obj3 = obj2;

}   // life of obj2 and obj3 ends here
```

// what is the necessary action to be taken ? Any guess ?

Hint : we have allocated a resource dynamically using a constructor.

Destructor – when & what?

```
class myRTOS  {
  private:
    int       reg_no;
    char  *ptr_name;

  public:
    myRTOS() ;     // default constructor
    myRTOS(char *s, int num);  // parameterized
    myRTOS(const myRTOS &ob); // copy con

    ~myRTOS();    // destructor
};

myRTOS :: ~myRTOS()
{
              delete [ ] ptr_name;
}


    ..............................
```

```
{
      myRTOS obj2(“Ajaykumar”,2);

      myRTOS obj3 = obj2;

}   // life of obj2 and obj3 ends here
```

// what is the necessary action to be taken ? Any guess ?

Hint : we have allocated a resource dynamically using a constructor.

Solution : Release the dynamically allocated memory

From Bird’s eye

- **Constructor**
  - Invoked when an object is created
  - Used to initialize object’s data
  - Used to allocate resources external to the program

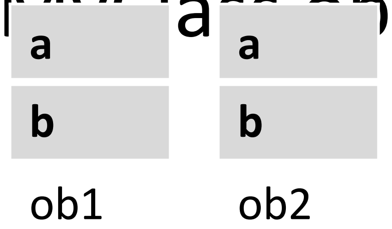
- **Destructor**
  - Invoked when an object goes out of scope
  - Used to de-allocate / release any resources allocated dynamically

static

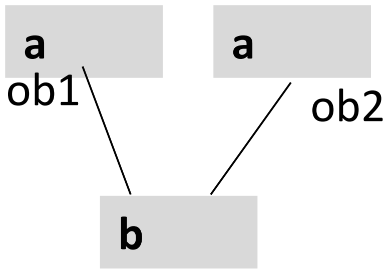
- In C++, static can be applied
  - Data members of a class
  - Member functions as well

Non-static vs static members

```
class MyClass {
    int a;
    int b;
};


MyClass ob1, ob2;
```



```
class MyClass {
    int a;
    static int b;
};


MyClass ob1, ob2;
```



static data

- Static data members act as global variables (for a class of objects)
- So each instance of the class has access to the same static data member.
- better than making data member a fully global variable because a static data member can be made private whereas a global cannot

Important – Defining static data member

- Static data members need an explicit definition
  - Class is a logical construct
  - static Within class is just declaration
  - Definition is needed for memory allocation
- We must provide a **definition** outside class

```
class MyClass {
   static int a;      // declaration
};
int MyClass :: a = 12;     // definition
                          // default value is 0
```

static and public

```
class A {
  public:
     static int
  a;
     int b;
};
int A :: a =
  10;
```

```
  A ob1, ob2;

  ob1.a = 20; // valid


Accessing public static
  member

  cout << A:: a ;
```

static data – a designers choice …

Counting number of objects

```
class ObjectCounter {
  public:
    static int count;

    ObjectCounter()
    {
      count++;
    }

    ~ObjectCounter()
    {
      count--;
    }
};
int
ObjectCounter::count;
```

```
cout <<  ObjectCounter :: count ;
```

applying common value among objects

```
class Bank_Account {
private:
    int acc_no;
    char *name;
    double amount;
    int period;

 static float rate_of_interest;
….
};

float Bank_Account ::
        rate_of_interest = 8.5;
```

static member function

```
class Bank_Account {
private:
    int acc_no;
    char *name;
    double amount;
    int period;

 static float rate_of_interest;
 void changeRate(float nrate);
….
};

float Bank_Account ::
        rate_of_interest = 8.5;
-----


BankAccount Ajay, Asha;

Ajay.changeRate()?
Asha.changeRate()?
```

```
BankAccount Ajay, Asha;

Ajay.changeRate()?
Asha.changeRate()?
```

Requirement :

We should be able to manipulate
Static data of class
independent of objects

static member function

```
class Bank_Account {
private:
    int acc_no;
    char *name;
    double amount;
    int period;
public:
 static float rate_of_interest;
static void
       changeRate(float nrate);
….
};

float Bank_Account ::
        rate_of_interest = 8.5;
-----
void Bank_Account ::
changeRate(float nrate)
{
     rate_of_interest = nrate;
}
```

```
BankAccount Ajay, Asha;

changeRate(9.2); ???// invalid

Bank_Account ::changeRate(9.2);
```

Requirement :
We should be able to manipulate Static data of class independent of objects

-They are independent of the objects
-Don't receive *this* pointer
(no object to invoke = no this pointer ☺)
- Can access only static members of a class

const

- Data member of a class
- Member function of a class
- Object also ☺ !!

- ***mutable*** – at your service!!

```
class MyClass
{
private:
        const int i;
        int j;
public:
        MyClass():i(10)
        {
                //i = 10;
                j = 20;
        }
```

Initializer List

```
MyClass(int a, int b)
             : i(a), j(b)
{


}
```

Note: intializer list can be used to initialize non const members also

const member function

```
class Bank_Account {
private:
    int acc_no;
    double amount;
public:
   void displayBalance() const;


….
};

void displayBalance() const
{
    //amount = amount - 100;
}
```

-const member function cannot modify any data

- known as accessor / read only function

-**Note** : all the show / display / get functions are usually defined as const members

-**Note** : keyword *const* is used in declaration and definition both

const object

- All the data of a const object becomes constant automatically
- Const object can access only const member functions

- Example

  const MyClass ob1; // just similar to const int a??

```
 class MyClass
 {
 private:
mutable int i;
        int j;
 public:
        MyClass();
        MyClass(int,int);
        void
        do_something()const;
 ……
 };
```

```
void MyClass ::
      do_something()const
{
      i  = 10;
}
----------

const MyClass obj1(1,1);

obj1.do_something();
```

```
class BCharacterAttribute : public BRoot {

public:
    BCharacterAttribute();
    BCharacterAttribute(const BCharacterAttribute &attr);
    void init();

public:
    static const unsigned int WIDTH_REGULAR;   // for extentionRatio
    static const unsigned int WIDTH_EXTENDED;

........
};
```

Function overloading

- Enables us to define more than one function with the same name
  - Yesterday we defined constructor functions

Example:

```
Class BankAccount {
private:
   // data goes here
public:
   void withdrawCash(long card_no, int
  pin);
   void withDrawCash(int acc_no, string
  signature);
};
```

Compilers view

- Overloaded functions can be differentiated if they are different in
  - Number of arguments
  - Types of arguments
  - Order of arguments
  - Const-ness of member functions

- However, I (compiler) ***cannot distinguish*** between

```
void do_something(int );
int do_something(int );
```

functions different in only their return type are not overloaded.

Beware of ambiguity

```
void doubleNumber(int x)
{
}

void doubleNumber(int &x)
{
}
-------------
int a = 10;
doubleNumber(a);  // ?????
```

```
void doSomething( )
{
}

void doSomething(int x= 1)
{
}
-------------
doSomething(); // ???????
```

Operator Overloading

- Most operators operate only on built-in data types.

```
int a = 10;
++a; // this is ok
MyCounter obj1;
++obj1; // what do I do with this object of user defined
type?
```

- Operator Overloading lets a developer add extra meaning to existing operators so that they can work with classes as well.

How to overload operator

- To overload an operator, an ***operator*** function is written
- An operator function is written either as

a member function.

or

friend function

Syntax :

return_type operator # (parameter list);

Unary operators !!

```
class MyCounter {
  private:
    int count;

  public:
    counter();
    counter(int ct);
    void display() const;

    MyCounter& operator ++();
    MyCounter operator ++(int);
};

MyCounter& MyCounter ::
                 operator++()  {
    count++;
    return *this;
}
```

```
// postfix
MyCounter MyCounter ::
             operator(int a){
    MyCounter ob;
    ob.count = count;
    ++count;
    return ob;
}

---------------
```

Operator function is invoked only when expression involves an object ☺

```
MyCounter cob1(10);

++cob1;
cob1++;

int I = 10;
++I;
I++;
```

friend in need!

```
class MyCounter {
  private:
    int count;

  public:
    counter();
    counter(int ct);
    void display() const;
  friend MyCounter
       operator++(MyCounter &);

  friend MyCounter
  operator ++(MyCounter &, int);

};
// Global operator fun
MyCounter
       operator++(MyCounter &ob)
{
    ob.count++;
    return ob; // no this here
}
```

```
// postfix
MyCounter
  operator++(MyCounter &ob,int
a){
    MyCounter loc1;
    loc1.count = ob.count;
    ++ob.count;
    return loc1;
}
```

Operator function is invoked only when expression involves an object ☺

```
MyCounter cob1(10);

++cob1;
cob1++;

int I = 10;
++I;
I++;
```

Binary operators

```
class MyCounter {
  private:
    int count;

  public:
    counter();
    counter(int ct);
    void display() const;
    MyCounter
        operator+(MyCounter &);

friend MyCounter operator -
     (MyCounter &, MyCounter &);
};

MyCounter
       operator +(MyCounter &ob)
{
       MyCounter loc;
       loc.count =
            this->count + ob.count;
       return loc;
```

```
// postfix
MyCounter operator-
 (MyCounter &ob1,MyCounter &ob2)
    MyCounter loc;
    loc.count =
           ob1.count - ob2.count;
     return loc;
}

-------------

MyCounter cob1(10);
MyCounter cob2(3);

MyCounter cob3;
cob3 = cob1 + cob2;

cob3 = cob1 - cob2;
```

= operator, special need

```
class myRTOS  {
  private:
    int       reg_no;
    char  *ptr_name;

  public:
    myRTOS() ;     // default constructor
    myRTOS(char *s, int num);  // parameterized
    myRTOS(const myRTOS &ob); // copy con
};

myRTOS :: myRTOS( const  myRTOS  &ob )
{
                this->reg_no = ob.reg_no;
                ptr_name =
                 new char[strlen(ob.ptr_name)+1];
                strcpy(ptr_name, ob.ptr_name);
}


   ............................
```

```
myRTOS obj2("Ajaykumar",2);

//Shallow copy solved using copy cons
myRTOS obj3 = obj2;

myRTOS obj4;
```

obj4 = obj3;

// default assignment operator is not useful here 'coz of the member wise assignment

Solution : write your own
operator = () ☺

= operator, special - how

```
class myRTOS  {
  private:
    int       reg_no;
    char  *ptr_name;

  public:
    myRTOS() ;     // default constructor
    myRTOS(char *s, int num);  // parameterized
    myRTOS(const myRTOS &ob); // copy con
};

myRTOS myRTOS :: operator =
                                    (const  myRTOS  &ob )
{
   this->reg_no = ob.reg_no;
   if(strlen(ptr_name) < strlen(ob.ptr_name)) {
               delete [] ptr_name;
               ptr_name = new char [strlen…….
               strcpy(ptr_name, ob.ptr_name);
   } else
               strcpy(ptr_name, ob.ptr_name);
}
```

```
myRTOS obj2("Ajaykumar",2);

//Shallow copy solved using copy cons
myRTOS obj3 = obj2;

myRTOS obj4;

obj4  = obj3;
```

Solution : write your own
operator = () ☺

// now instead of default member wise assignment, our ***intelligent*** assignment is used

Overloading new & delete

For Single object

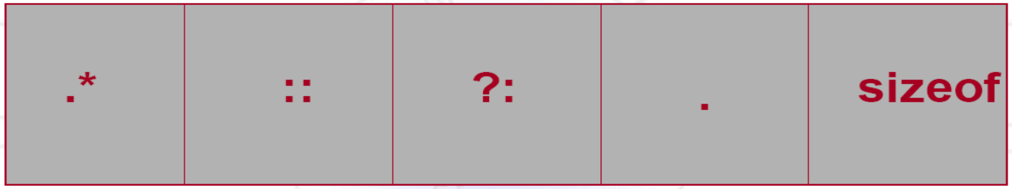
```
class MyClass {
public:
void *operator new(size_t size);
void operator delete(void *p);
};

MyClass *mptr = new MyClass;
…….
delete mptr;
```

For Array

```
class MyClass {

public:

void *operator new[ ](size_t size);

void operator delete[ ](void *p);

};


MyClass * mptr = new MyClass [5];

……..

delete [ ] mptr;
```

Some Important points

- Operator overloading is just another way of making a function call ☺
- Rules of overloading operators
  - Cannot change the number of operands an operator requires
  - Cannot change the precedence of operators
  - Cannot create any new operators
  - Following operators cannot be overloaded



.*	::	?:	.	sizeof

    - typeid(), static_cast, dynamic_cast, const_cast, reinterpret_cast
  - = , ->, [ ] and () must be overloaded using non-static member functions
  - Insertion (<<) and extraction (>>) must be overloaded as ***friend***

Day 2

- Inheritance Basics
- Role of mode of inheritance
- Types of inheritance
- A word on constructors
- Hybrid inheritance and virtual base class

Inheritance

“the mechanism by which one class acquires the properties of another class”

Why Inheritance ?

Sometimes

We don’t have access to existing class’ source code!!

Someone’s debugging efforts might go waste after your modifications

Inheritance

- Base Class (or superclass):

  The class being inherited from

- Derived Class (or subclass):

  The class that inherits

Syntax of derivation:

```
class base_class_name
{

};
```

keyword

Mode of derivation

```
class derived_class_name : mode base_class_name

{

        body of the derived class

};
```

Base class visibility	Mode of inheritance		
	public	private	protected
private	Not Accessible	Not Accessible	Not Accessible
protected	Protected	Private	Protected
public	Public	Private	protected

protected:

protected members are accessible by the member functions of its own class and by any class ***derived*** **from it.**

Types of inheritance

1. Single inheritance

2. Multiple inheritance

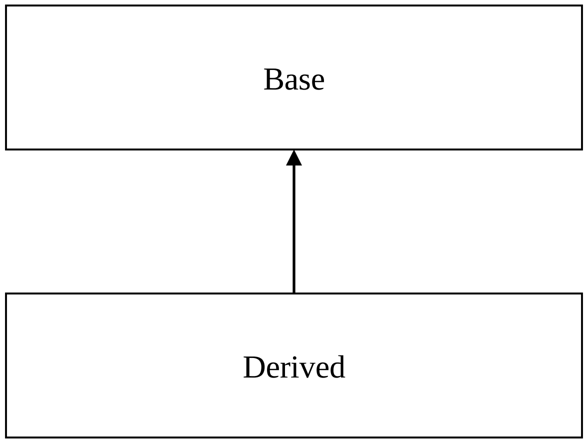
3. Multilevel inheritance

4. Hierarchical inheritance

5. Hybrid inheritance

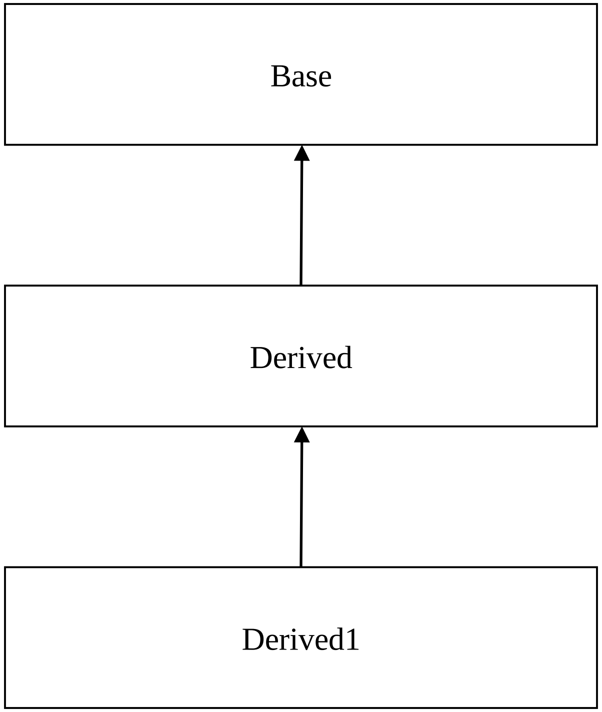
single inheritance

A derived class with one base class.



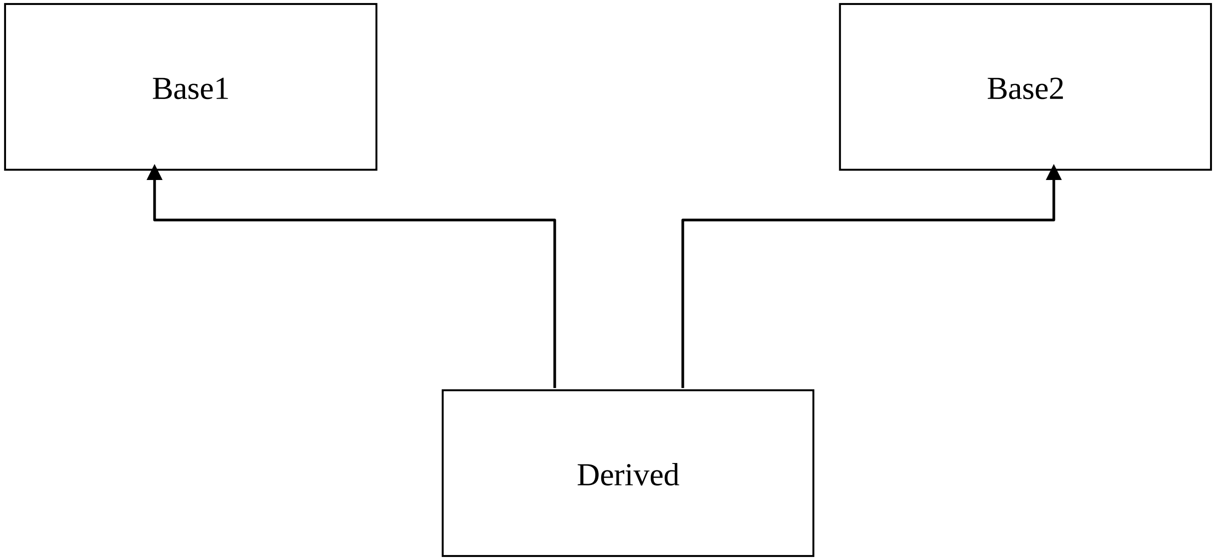
Multi Level Inheritance

Deriving a new class from another derived class.



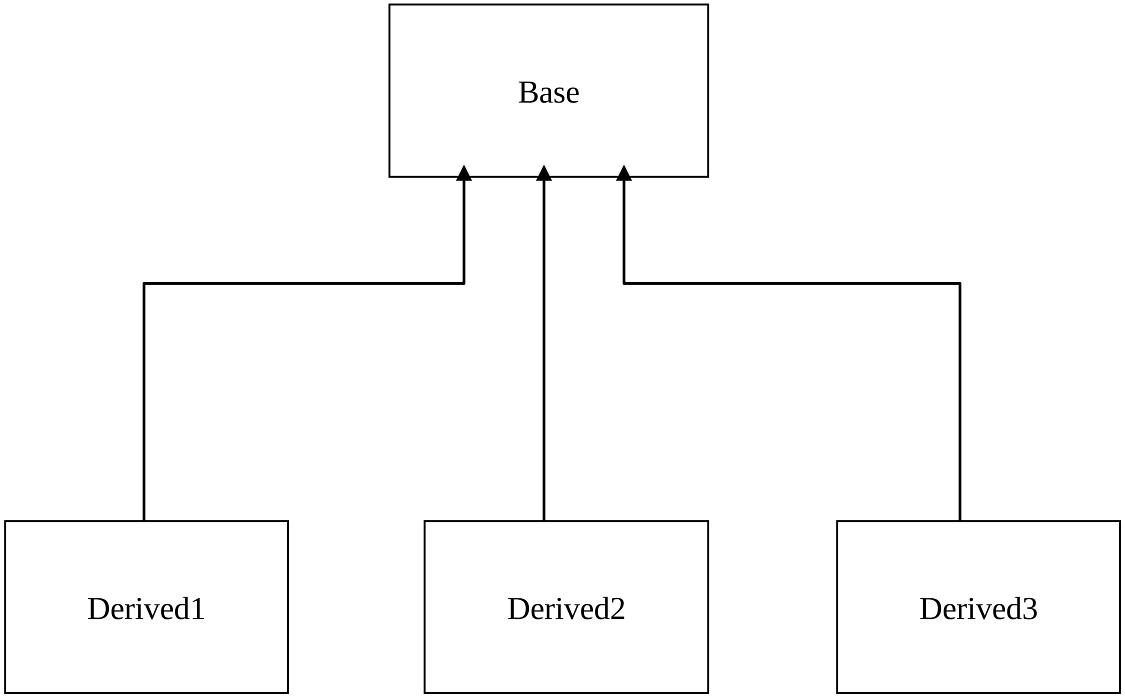
Multiple inheritance

A derived class with more than one base class



Hierarchical inheritance

one base class inherits more than one derived class.



Constructors in Inheritance

1. When both the derived class and the base class contain constructors, the base class constructor is executed first and then the derived class constructor.

2. In case of multiple inheritance the base classes are constructed in the order in which they appear in the declaration of the derived class and the constructors will be executed in the order of inheritance.

```
class Derived : public Base1, public Base2 {
 };
```

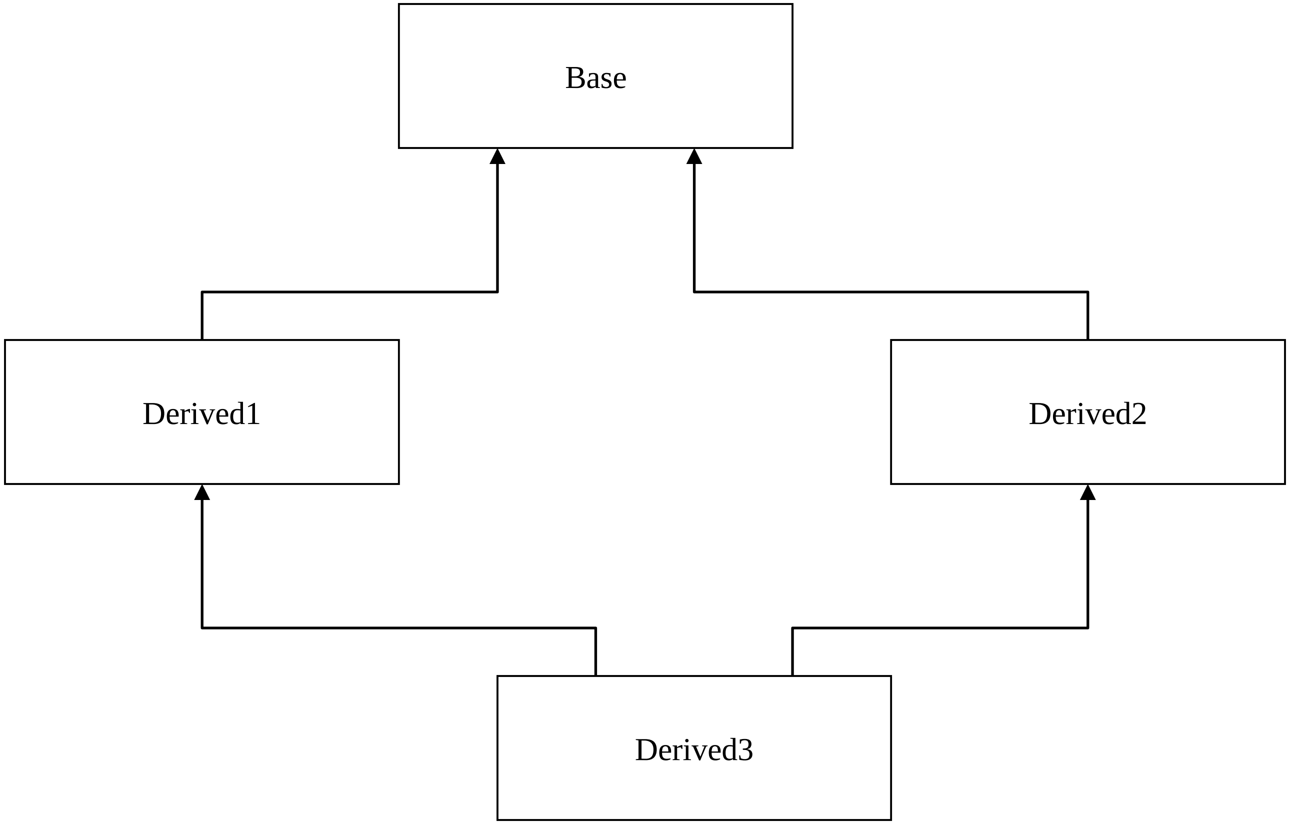
Parameterized Constructors

1. **If the base class contains the parameterized constructor, then the derived class must also have a constructor and pass the arguments to the base class constructor.**

2. Since the derived class takes the responsibility of supplying the initial values to the base class. The initial values are supplied when a derived class object is declared.

3. The constructor of the derived class receives the entire list of values as its arguments and passes them to base class constructor in the order in which the base class constructors expects. After that, the derived class constructor is executed.

Hybrid inheritance

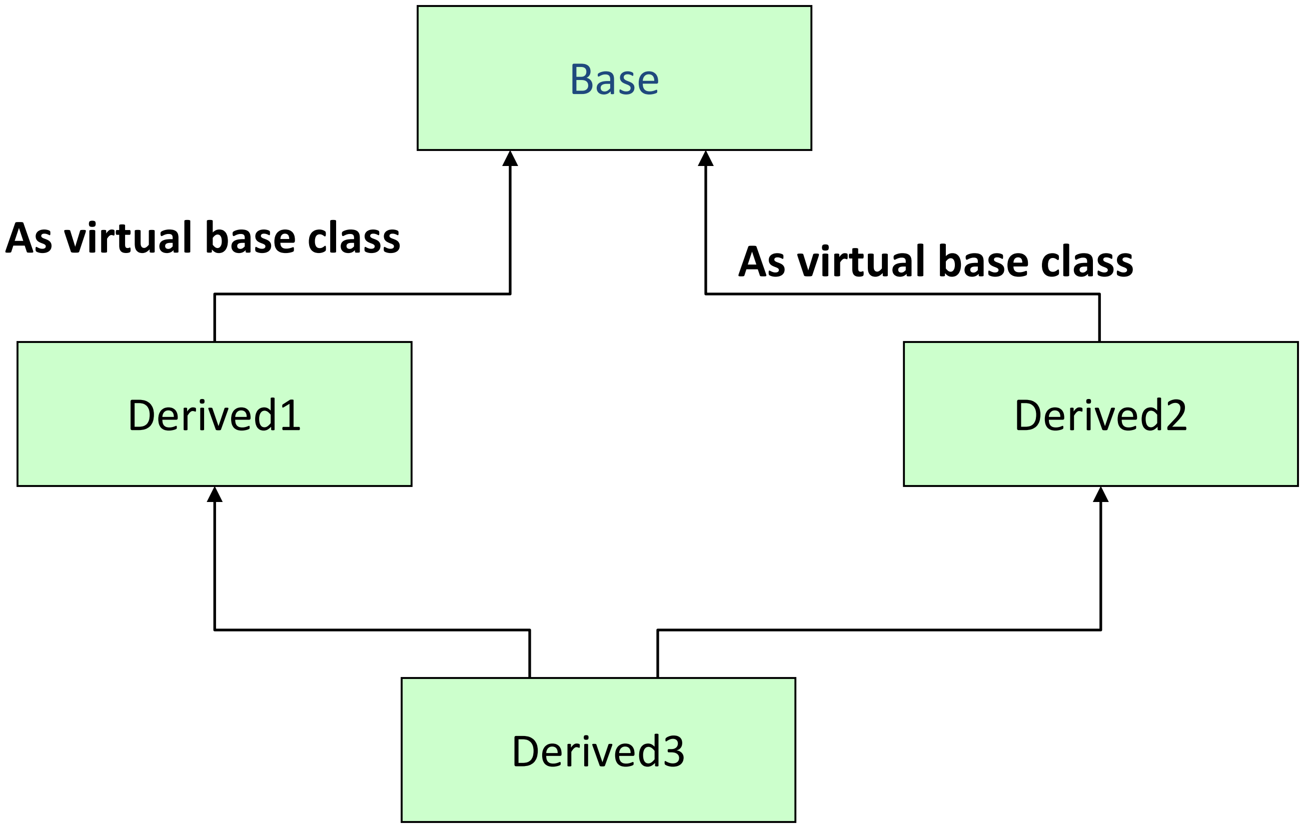


virtual base class - need

```
class Top {
        public: int a;
};
class Left : public Top {
        public: int b;
};
class Right : public Top {
        public: int c;
};
```

```
class Bottom : public Left,
   public Right
{
        public: int d;
};
---------------
Bottom obj;
obj.a  = 100;
// Which a
```

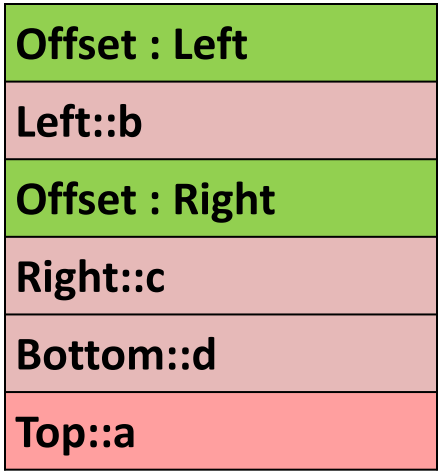
virtual base class



virtual base class : object memory layout

```
class Top {
        public: int a;
};
class Left : virtual public
   Top {
        public: int b;
};
class Right : public virtual
   Top {
        public: int c;
};
```

```
class Bottom : public Left, public
     Right
{
            public: int d;
};
```



Offset : Left

Left::b

Offset : Right

Right::c

Bottom::d

Top::a

Layout of object of Bottom

Execution of class constructors

Method of inheritance	Order of execution

class derived: public base	base()
	derived()

class derived: public base1, public base2	base1()
	base2()
	derived()

class derived: public base1, virtual public base2	base2()
	base1()
	derived()

Virtual Functions

- Function overriding
- Upcasting
- Need of virtual function
- How it works?
- virtual destructor .. why?

Function overriding

```
class Computer {
protected:
        double price;
public:
   Computer();
   Computer(double);
  double CalculatePrice();
};
```

```
class Laptop: public
   Computer {
// laptop memebrs declared
   here
};
--------------------------------
   -----
Laptop  lob;
lob.CalculatePrice();
```

Override the base class function

```
class Computer {
protected:
        double price;
public:
   Computer();
   Computer(double);
  double CalculatePrice();
};
```

```
class Laptop: public Computer  {
// laptop memebrs declared here
double CalculatePrice();
};
------------------------------
Laptop  lob;
lob.CalculatePrice();
// object type determines the
   definition
```

Caution : while overriding, keep the parameter list same

Upcasting

```
class Computer {
protected:
       double price;
public:
   Computer();
   Computer(double);
  double CalculatePrice();
};
class Laptop: public
   Computer {           };
```

```
Computer * cPtr;

Computer cob;

cPtr = &cob; // perfect

Laptop  lob, *lPtr;

cPtr = &lob;

cob = lob; // ok

//downcasting not allowed

lPtr = &cob;

lob = cob;
```

`Try safe downcasting with static_cast / dynamic_cast`

using overriding & upcasting together

```
class Computer {
protected:
   double price;
public:
     double CalculatePrice();
};
class Laptop: public Computer
{    public:
     double CalculatePrice();
};
```

```
Computer * cPtr;

Computer cob;

cPtr = &cob; // perfect

cPtr->CalculatePrice();


Laptop  lob;

cPtr = &lob;

cPtr->CalculatePrice();
```

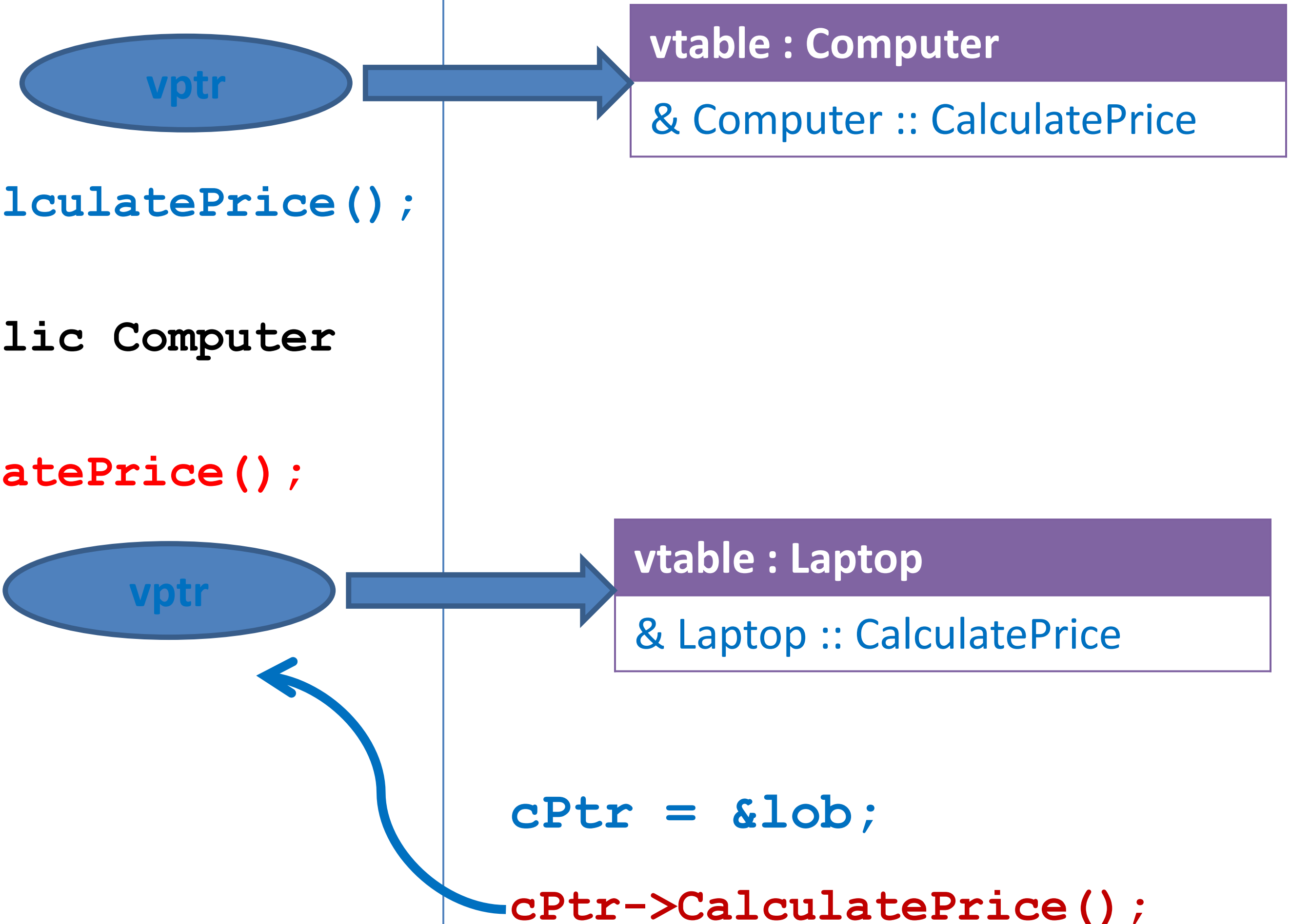
using overriding & upcasting together

```
class Computer {
protected:
   double price;
public:
virtual double CalculatePrice();
};
class Laptop: public Computer
{  public:
     double CalculatePrice();
};
```

```
Computer * cPtr;
Computer cob;
cPtr = &cob; // perfect
cPtr->CalculatePrice();


Laptop  lob;
cPtr = &lob;


cPtr->CalculatePrice();
```

vtable & vptr – virtual world ☺

```
class Computer {
protected:
   double price;
public:
virtual double CalculatePrice();
};
class Laptop: public Computer
{  public:
     double CalculatePrice();
};
```



```
cPtr = &lob;
cPtr->CalculatePrice();
```

Creating derived objects dynamically?

```
class Computer {
public:
   Computer();
   ~Computer();
};
class Laptop: public Computer
{   public:
        Laptop();
        ~Laptop();
};
```

```
Computer * cPtr;

cPtr = new Laptop;

----

delete cPtr;
```

order:

Computer()

Laptop()

~Computer()

Destructor of Derived class is not invoked

Solution: declare base destructor as virtual

```
class Computer {
public:
   Computer();
   virtual ~Computer();
};
class Laptop: public Computer
{   public:
        Laptop();
        ~Laptop();
};
```

```
Computer * cPtr;

cPtr = new Laptop;

----

delete cPtr;
```

```
order:
Computer()
Laptop()
~Laptop()
~Computer()
```

Destructor of Derived class is invoked now

Pure virtual function & abstract class

```
class Shape {
   private:
        LineStyle style;
        LineColor color;
        FillColor color;
   public:
   virtual void Draw()
   {
        // empty?
   }
};
```

```
Class Circle:public Shape {

public:

   void Draw()        {

        I know how to!

   }

};

class Rectangle : public

   Shape {

public:

   void draw () {

        // I know how to

   }

};
```

```
Shape Obj;
Obj.draw();   // ???????????
```

Pure virtual function & abstract class

```
class Shape {
   private:
         LineStyle style;
         LineColor color;
         FillColor color;
   public:
   virtual void Draw() = 0;
};
```

Shape is now Abstract Class

Shape Obj;   // not allowed

```
Class Circle:public Shape {
public:
   void Draw()       {
         I know how to!
   }
};
```

Derived class must define Draw()

```
class Rectangle : public
   Shape {
public:
   void draw () {
         // I know how to
   }
};
```

Abstract Class

- class that contains at least one ***pure virtual function***
- vtable created, but incomplete
- Cannot be instantiated , no objects
- Derived classes must provide definition for ***that*** function

- **Use**: **Abstract Base class** should hold all the common attributes and operations (*virtual / pure virtual*) which can be inherited.
  - Advantage : uniformity in interface of derived classes

Templates

Templates

- **Templates**
  - **Generate a function or class**
- **True polymorphism**
  - **Choice of which function to execute is made during run time**
  - **Allows the creation of generic functions and classes**
  - **The type of data upon which the function or class operates is specified as a parameter**
- Advantage: We can use a function or class template with many datatypes ☺

Function Templates

- Describes a function format
- when instantiated with particular datatype, generates a function definition

```
template <class T>
return-type function-name (parameter list)
{
        // body of function
}
```

Write once, use multiple time

Function Templates

```
template <class T>
return-type function-name (parameter list)
 {
        // body of function
 }
```

- “T” is a placeholder for a data type used by the function
- Name of the data type is passed in the function call, implicitly ☺
- **Compiler generates the actual definition**

Example

- The code below tells the compiler two things:

```
template <class T>
void swapargs(T &a, T &b)
{
          T temp;
         temp = a;
         a = b;
         b = temp;
}
```

  - That a template is being created
  - That a generic definition is beginning
- T is a generic type that is used as a placeholder for the types to be eventually used: int, float, char

An Example Function Template

Indicates a template is being defined

Indicates T is our formal template parameter

```
template <class T>
void swapargs(T &a, T &b)
{
   T temp;
   temp = a;
   a = b;
   b = temp;
}
```

What Compiler does here?

- Code segment

```
int i = 10, j = 20;
...
swapargs(i, j);
```

- Causes the following function to be generated from our template

```
void swapargs(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

swapargs( ) Template

- Code segment

```
char a = ‘x’, b = ‘z’;
...
swapargs(a, b);
```

- Causes the following function to be generated from our template

```
void swapargs(char &a, char &b)
{
    char temp;
    temp = a;
    a = b;
    b = temp;
}
```

Templates - Important Terms

- A “template function” is also called a “generic function”
- When the compiler creates a specific version of this function, it is said to have created a “specialization.”
- A “specialization” is also called a “generated function.”
- The act of generating a function is referred to as “instantiating” it.
- “A generated function is a specific instance of a template function.”

Functions with Multiple Generic Types

```
template <class T1, class T2>
void myfunc(T1 x, T2 y)
{
        cout << x << " " << y << endl;
}
int main()
{
   myfunc(10, "hi");
   myfunc(0.23, 10L);
 }
```

Class templates

```
template <class T>
class myclass
{
     T values [2];
  public:
     //parameterized constructor
     myclass (T first, T second)
     {
          values[0]=first;
          values[1]=second;
     }
};
```

The class that we have just defined serves to store two elements of any valid type**.**

For example:

if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
Myclass < int > myobject (115, 36);
```

this same class would serve also to create an object to store any other type:

```
myclass<float> myfloats (3.0, 2.18);
```

```
// class templates
#include <iostream.h>

template <class T>
class myclass
{
     T value1, value2;
 public:
     myclass (T first, T second)
     {
          value1=first;
          value2=second;
     }
     T getmax ();
};
```

Defining outside!

```
template <class T>
T myclass<T>::getmax ()
{
  T retval;
  retval = value1>value2? value1 : value2;
  return retval;
}
// Member function of a template class is
  also a template function itself ☺
int main ()
{
  myclass <int> myobject(100, 75);

   cout << myobject.getmax();
  return 0;
}
```

For templates

- Declaration and definition in the same file ?????
- Remember : member function of a template class is also a template function

- Advantages
  - True Reusability
  - Generic nature: works with (M)any types of data!
  - Saves development time.
  - Write once, use many times ☺

Exception Handling

Exception Handling

- Exceptions
  - Indicate problems that occur during a program's execution
  - Occur infrequently
- Exception handling
  - Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - Makes programs robust and fault-tolerant

Fundamental Philosophy

- Mechanism for sending an exception signal up the call stack
    - Regardless of intervening calls

- Note: there is a mechanism based on same philosophy in *C*
    - **`setjmp(), longjmp()`**
    - See man pages

Fundamental Philosophy (continued)

- Programs can
  - Recover from exceptions
  - Hide exceptions
  - Pass exceptions up the “chain of command”
  - Ignore certain exceptions and let someone else handle them

`try` Blocks

- Keyword **`try`** followed by braces (**`{}`**)
- Should enclose
  - Statements that might cause exceptions
  - Statements that should be skipped in case of an exception

Software Engineering Observation

- Exceptions may surface
  - through explicitly mentioned code in a **`try`** block,
  - through calls to other functions and
  - through deeply nested function calls initiated by code in a **`try`** block.

Example

- First Example

```
#include <iostream>
using namespace std;
int main ()
{
          try
          {
            throw 10;
          }
          catch (int e)
          {
            cout  << “We have a problem!!” << endl;
          }
          return 0;
 }
```

```
Output : We have a problem!!!
```

`Catch` Handlers

- Immediately follow a **`try`** block
  - One or more **`catch`** handlers for each **`try`** block
- Keyword **`catch`**
- Exception parameter enclosed in parentheses
  - Represents the type of exception to process
  - Can provide an optional parameter name to interact with the caught exception object
- Executes if exception parameter type matches the exception thrown in the **`try`** block
  - Could be a base class of the thrown exception's class

Catch Handlers (continued)

```
try {
   // code to try
}
catch (exceptionClass1 &name1) {
   // handle exceptions of exceptionClass1
}
catch (exceptionClass2 &name2) {
   // handle exceptions of exceptionClass2
}
catch (exceptionClass3 &name3) {
   // handle exceptions of exceptionClass3
}
...
/* code to execute if
        no exception or
        catch handler handled ex
```

All other classes of exceptions are not handled here

catch clauses attempted in order; first match wins!

Stack Unwinding

- Occurs when a thrown exception is not caught *in a particular scope*
- *Unwinding a Function* terminates that function
  - All local variables of the function are destroyed
    - Invokes destructors
  - Control returns to point where function was invoked
- Attempts are made to catch the exception in outer **`try`**…**`catch`** blocks
- If the exception is never caught, the function **`terminate`** is called

Throwing an Exception

- Use keyword **`throw`** followed by an operand representing the type of exception
  - The **`throw`** operand can be of any type
  - If the **`throw`** operand is an object, it is called an **`exception`** object
- The **`throw`** operand initializes the exception parameter in the matching **`catch`** handler, if one is found

When to Use Exception Handling

Don’t use for routine stuff such as end-of-file or null string checking

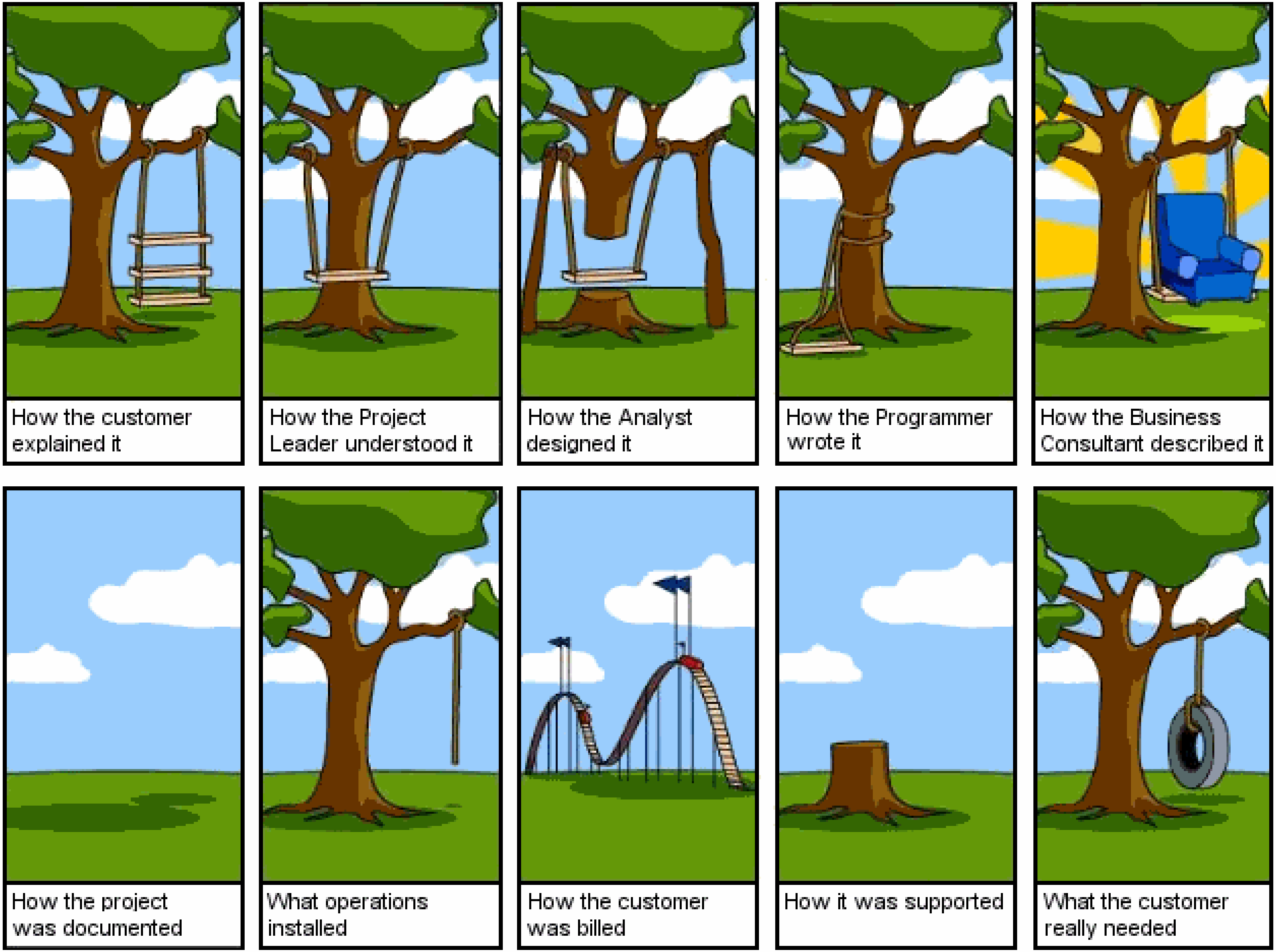
- To process synchronous errors
  - Occur when a statement executes
- Not to process asynchronous errors
  - Occur in parallel with, and independent of, program execution
- To process problems arising in predefined software elements
  - Such as predefined functions and classes
  - Error handling can be performed by the program code to be customized based on the application’s needs

Constructors and Destructors

- Exceptions and constructors
  - Exceptions enable constructors to report errors
    - Unable to return values
  - Exceptions thrown by constructors cause any already-constructed component objects to call their destructors
    - Only those objects that have already been constructed will be destructed
- Exceptions and destructors
  - Destructors are called for all automatic objects in the terminated **`try`** block when an exception is thrown
    - Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
  - If a destructor invoked by stack unwinding throws an exception, function **`terminate`** is called

Object Oriented Analysis & Design

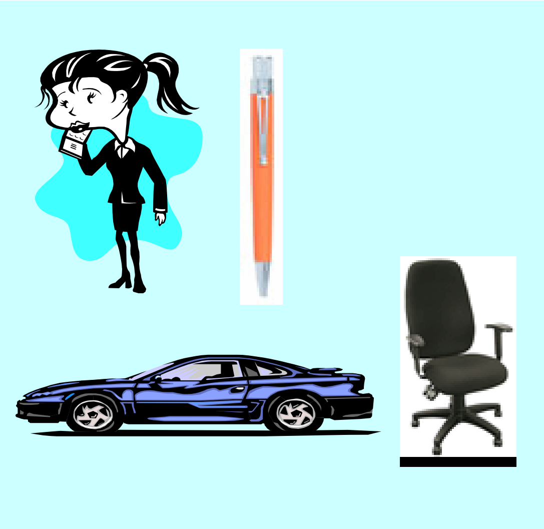
Why?


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

What is Object-Orientation?

- What is Object？

- An "object" is anything to which a concept applies, ***in our awareness***
- Things drawn from the problem domain or solution space.
  - E.g., a living person in the problem domain, a software component in the solution space.



- A structure that has identity and properties and behavior
- It is an instance of a collective concept, i.e., a class.

What is Object-Orientation

- Abstraction and Encapsulation

Abstraction

Focus on the essential

Omits tremendous amount of details

…Focus on what an object “is and does”

Encapsulation

a.k.a. information hiding

Objects encapsulate:

property

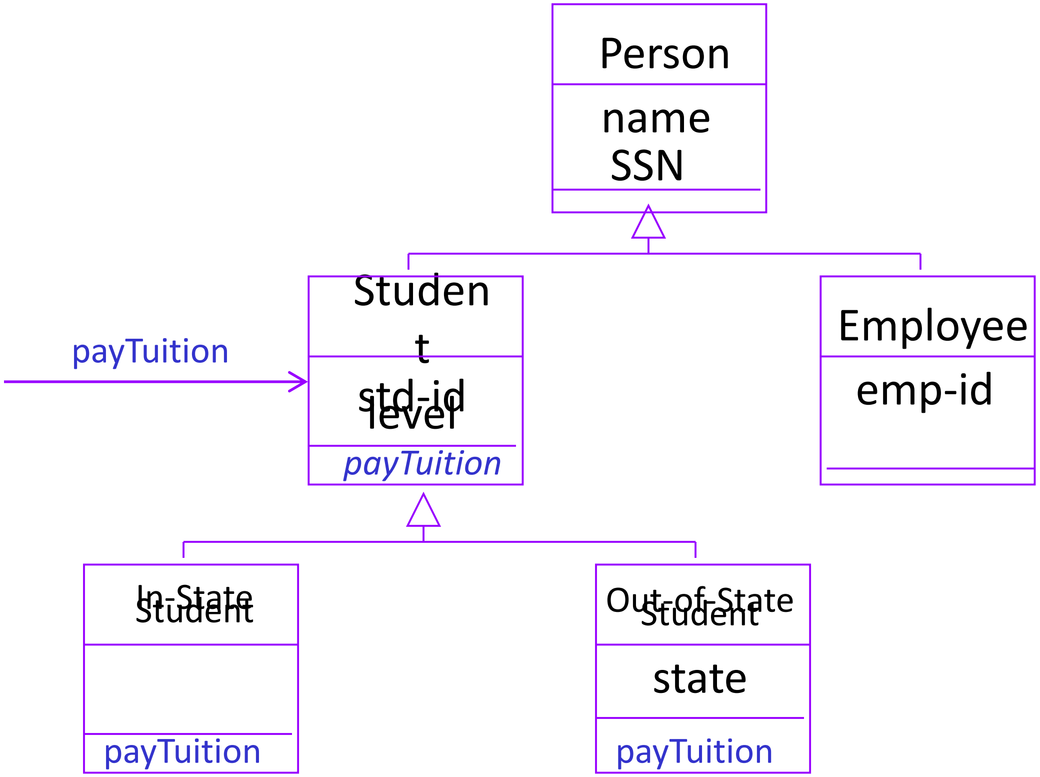
behavior as a collection of methods invoked by messages

…state as a collection of instance variables

What is Object-Orientation

- Polymorphism

Objects of different classes respond to the same message differently.

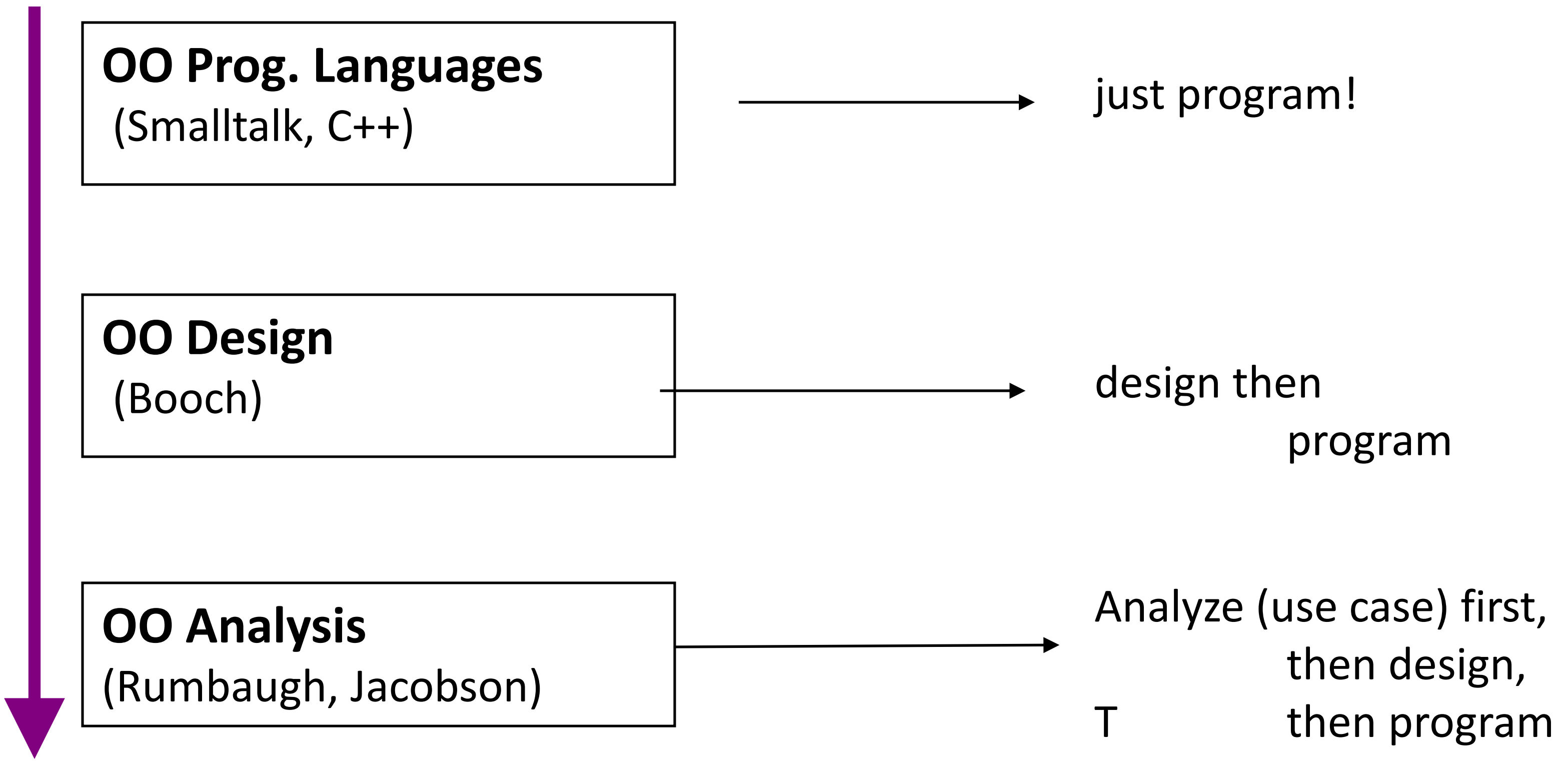


How to Do OOAD

- Historical Perspective

OO Technology

Process Perspective



OO Prog. Languages
(Smalltalk, C++)

just program!

OO Design
(Booch)

design then
program

OO Analysis
(Rumbaugh, Jacobson)

Analyze (use case) first,
then design,
T then program

Where are we heading?

Introduction to OOAD - Summary

Why

- Once Software Crisis due to Communication and Complexity
- Languages, Concepts, Models
- OO for Conceptual Modeling

What

- Fundamental OO Concepts
- A little taste of UML

How

- OO development processes & (Design) Patterns

What is UML?

- Standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems.
- The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
- The UML is a very important part of developing object oriented software and the software development process.
- The UML uses mostly graphical notations to express the design of software projects.
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Overview of UML Diagrams

Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

Interaction

: emphasize object interaction

- Communication(collaberation)
- Sequence
- *Interaction overview*
- *Timing*

Types of Diagrams

- Structural Diagrams – focus on static aspects of the software system
  - **Class**, Object, Component, Deployment

- Behavioral Diagrams – focus on dynamic aspects of the software system
  - **Use-case**, **Interaction**, State Chart, Activity

Behavioral Diagrams

- **Use Case Diagram** – high-level behaviors of the system, user goals, external entities: actors
- **Sequence Diagram** – focus on time ordering of messages
- **Collaboration Diagram** – focus on structural organization of objects and messages
- **State Chart Diagram** – event driven state changes of system
- **Activity Diagram** – flow of control between activities

High Level Design using Class Diagrams

Overview

- How class models are used? Perspectives
- Classes: attributes and operations
- Associations
  - Multiplicity
- Generalization and Inheritance
- Aggregation and composition

- Later: How to find classes
  - small and larger systems

Developing Class Models

- Class diagrams used for different purposes during different times in the development life-cycle
  - Models that are close to code

    vs.
  - Models that support earlier modeling:
    - For domain analysis
    - For requirements specification
- Class diagrams developed iteratively
  - Details added over time during lifecycle
  - Initially: missing names, multiplicities, other details

More Abstract Perspectives

- Some define particular <u>perspectives</u> for class models:
  - Conceptual
  - Specification
  - Implementation
- Conceptual perspective
  - Represents concepts in the domain
  - Drawn with no regard for implementation (language independent)
  - Used in requirements analysis
- Specification
  - Interfaces defined: a set of operations
  - Focus on interfaces not how implementation broken into classes
  - Sometimes known as a “type”

Design and Code Level Perspectives

- What’s useful at the design level?
  Your thoughts here:

Implementation Level Class Diagrams

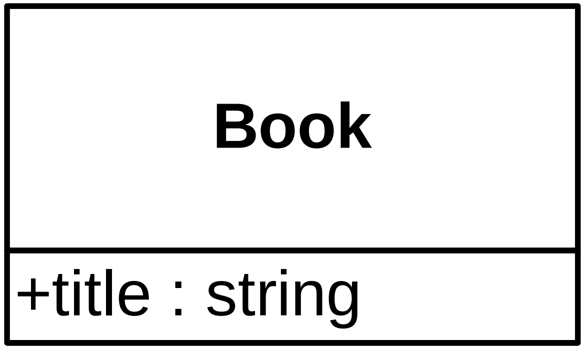
- Implementation
  - Direct code implementation of each class in the diagram
  - A blue-print for coding
- Practical issue: How much detail?
    - getters and setters?
    - library classes like String?
    - Reverse- and round-trip engineering tools

Documenting Your Objects

- Need some kind of record of your definitions
  - Your white-board?
  - A simple glossary
  - A *data dictionary* (perhaps in a CASE tool)
- What to define?
  - Attributes, operations for each class
  - Also relationships between classes
- Can you define classes of related objects?
  - Inheritance, Java interfaces

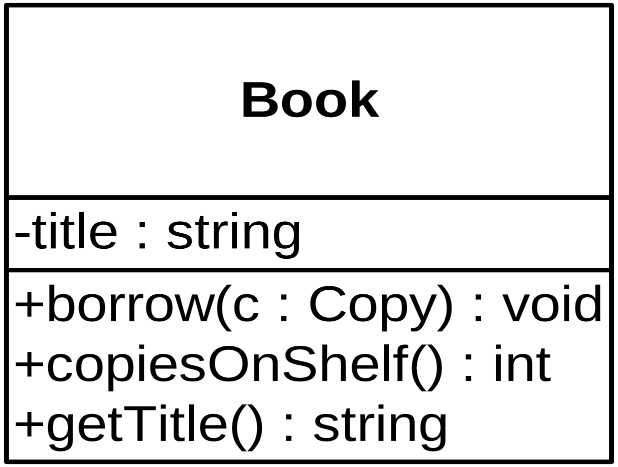
Classes in UML Diagrams

- Attributes in middle
- Operations at bottom
  - Can be suppressed. (What level of abstraction?)
- Attribute syntax:
  name : type = default
- Operation syntax:
  name (params) : return type
- Visibility
  + public
  - private
  # protected etc.
  nothing? Java’s default-package?



Book

+title : string



Book

-title : string

+borrow(c : Copy) : void
+copiesOnShelf() : int
+getTitle() : string

Associations

- For "real-world objects" is there an association between classes?
- Classes A and B are associated if:
  - An object of class A sends a message to an object of B
  - An object of class A creates an instance of class B
  - An object of class A has an attribute of type B or collections of objects of type B
  - An object of class A receives a message with an argument that is an instance of B (maybe…)
    - Will it "use" that argument?
- Does an object of class A need to know about some object of class B?

More on Associations

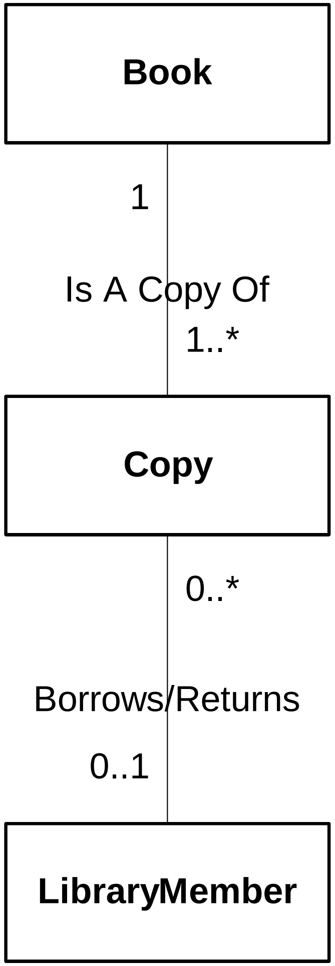
- Associations should model the reality of the domain and allow implementation
- Associations are between classes
  - A link connects two specific objects
  - Links are instances of associations
  - Note we could draw an object diagram to show objects and links
    - But often interaction diagrams are more useful for modeling objects
- Note: In practice, early in modeling, we may not name associations
- Note: One may choose to have a dynamic view associations: if at run-time two objects exchange messages, their classes must be associated

Multiplicity

- Also known as cardinality
- Objects from two classes are linked, but how many?
  - An exact number: indicated by the number
  - A range: two dots between a pair of numbers
  - An arbitrary number: indicated by * symbol
  - (Rare) A comma-separated list of ranges
- Examples:
  1   1..2   0..*   1..*   * (same as 0..* but…)
- Important: If class A has association X with class B
  - The number of B's for each A is written next to class B
  - Or, follow the association past the name and then read the multiplicity
- Implementing associations depends on multiplicity

Examples of Associations

- From a Library catalog example
- One book has 1 or more copies
- One copy is linked to exactly one book
- Should there be two associations: borrows and returns?
- One copy is borrowed by either zero or one LibraryMember



Generalization and Inheritance

- You may model “inheritance” early but not implement it
  - Generalization represents a relationship at the conceptual level
  - Inheritance is an implementation technique
- Generalization is just an association between classes
  - But so common we put a “triangle” at the superclass
- Note this is a relationship between classes
  - So no multiplicities are marked. Why not?
- Inheritance may not be appropriate when it’s time to implement
  - Objects should never change from one subclass to another
  - *Composition* can be used instead

Aggregation and Composition

- Again, just a specific kind of association between classes
  - An object of class A is part of an object of class B
  - A part-whole relationship
- Put a diamond on the end of the line next to the "whole"
  - Aggregation (hollow diamond): really no semantics about what this means!
  - Composition (solid diamond): a stronger relationship

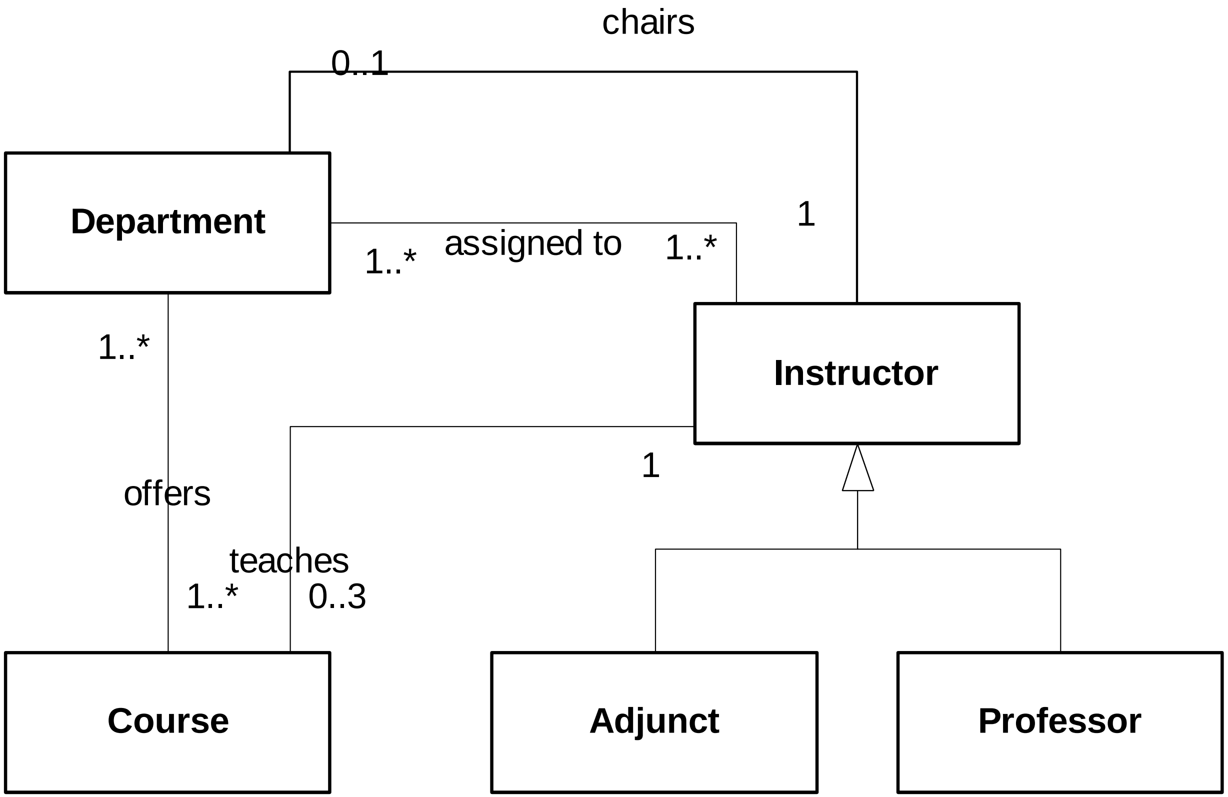
Aggregation and Composition (cont’d)

- Composition
  - The whole strongly owns the parts
  - Parts are copied (deleted, etc.) if the whole is copied (deleted, etc.)
  - A part cannot be part of more than one whole
  - Mnemonic: the stronger relationship is indicated by the stronger symbol (it’s solid)
- Aggregation and composition associations are not named
- They do have multiplicities
- They can be used too often. If in doubt, use a “plain”, named association.

Example 1: University Courses

- Some instructors are professors, while others have job title adjunct
- Departments offer many courses, but a course may be offered by >1 department
- Courses are taught by instructors, who may teach up to three courses
- Instructors are assigned to one (or more) departments
- One instructor also serves a department chair

Class Diagram for Univ. Courses



- Note this implies adjuncts can be chairs

Class Attributes, Operations

- Recall in Java and C++ you may have *class* attributes and *class* operations
  - keyword *static* used
  - One attribute for all members of class
  - An operation not encapsulated in each object, but “defined in” that class’ scope
- In UML class diagrams, list these in the class box’s compartments, but underline them

Dependencies

- Dependency: A using relationship between two classes
  - A change in the specification of one class may affect the other
  - But not necessarily the reverse
- Booch says: use dependencies not associations when one class uses another class as an argument in an operation.
- Often used for other things in UML: A general relationship between “things” in UML
  - Often use a stereotype to give more info
- Uses: binding C++ class to template; Java interfaces; a class only instantiates objects (a factory)

Stereotypes

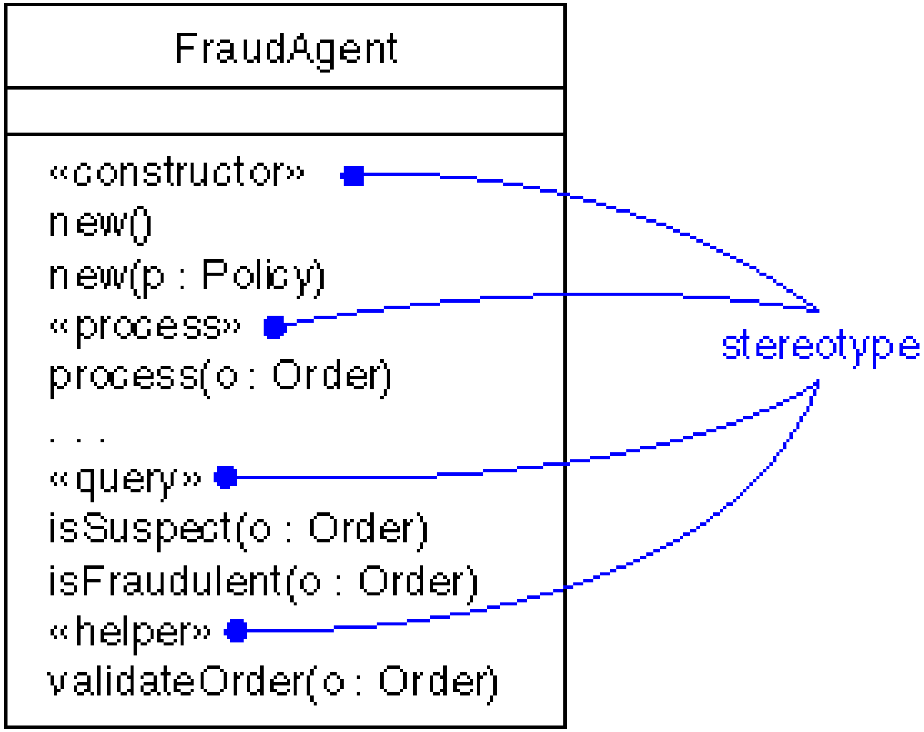
- Extends the “vocabulary” of UML
- Creates a new kind of building block
  - Derived from existing UML feature
  - But specific for current problem
- Also, some pre-defined stereotypes
- UML allows you to provide a new icon!
- Syntax: Above name add <<stereotype>> inside guillemets (French quotes)
- Again, used to provide extra info about the UML modeling construct

Stereotypes (cont'd)

- UML predefines many:
  - Classes: <<interface>>, <<type>>, <<implementationClass>>, <<enumeration>>, <<thread>>
  - Constraints: <<precondition>> etc.
  - Dependencies: <<friend>>, <<use>>
  - Comments: <<requirement>>, <<responsibility>>
  - Packages: <<system>>, <<subsystem>> (maybe classes, too)
- Or, create your own if needed.

Class Categories

- You can use stereotypes to organize things by category within a class box

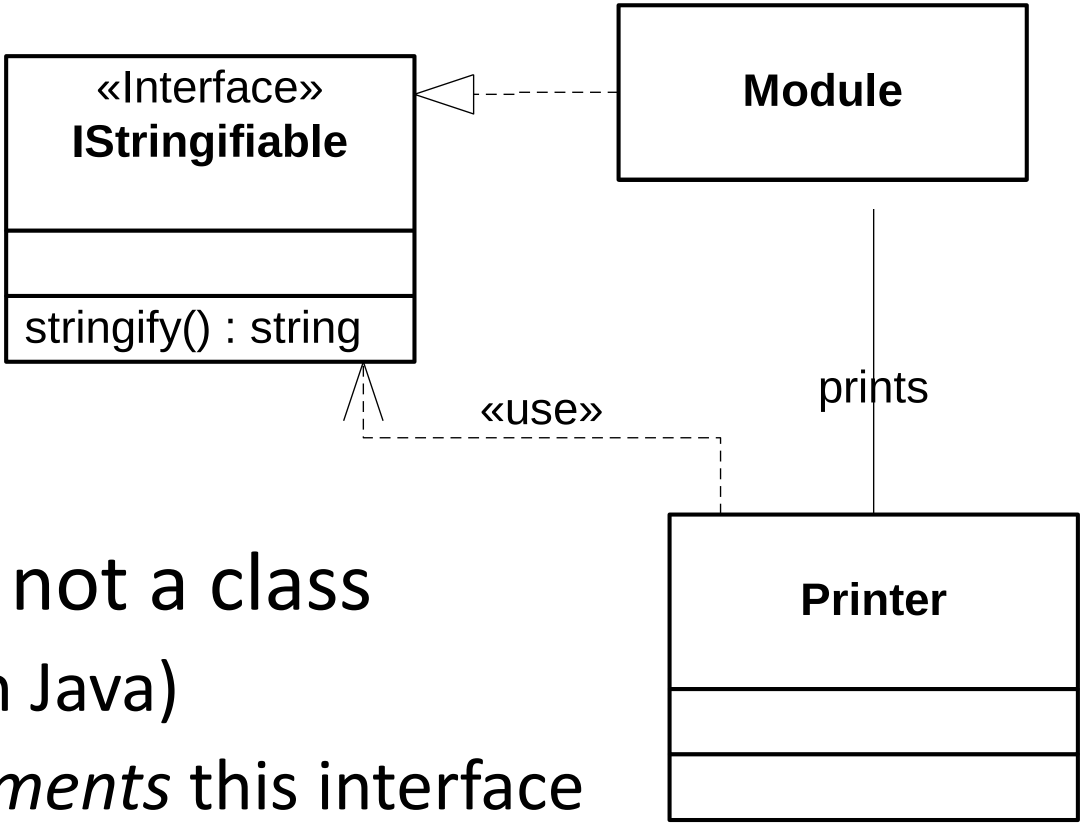


FraudAgent

«constructor»
new()
new(p : Policy)
«process»
process(o : Order)
. . .
«query»
isSuspect(o : Order)
isFraudulent(o : Order)
«helper»
validateOrder(o : Order)

stereotype

Stereotype Example



- IStringifiable is not a class
  - Interface (as in Java)
  - Module *implements* this interface
- Printer depends on what’s in the interface

Interfaces

- Interface: specifies a set of operations that any class *implementing* or *realizing* the interface must provide
  - More than one class may realize one interface
  - One class may realize more than one interface
  - No attributes, and no associations
- Notation:
  - Use <<interface>> with a class; list operations
  - "Lollipop" notation

Interface Example Diagram

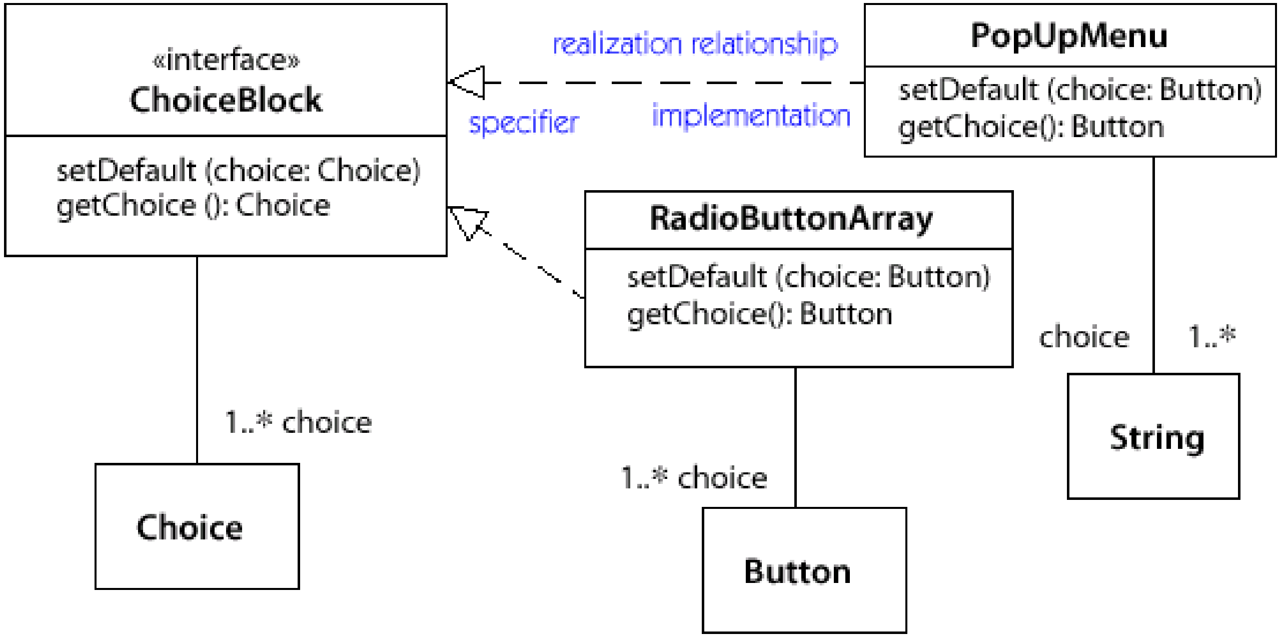


Figure 4-9. *Realization relationship*

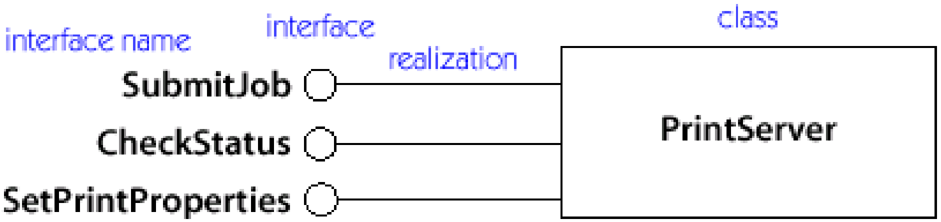


Figure 4-10. *Interface and realization icons*

Classes Realize an Interface

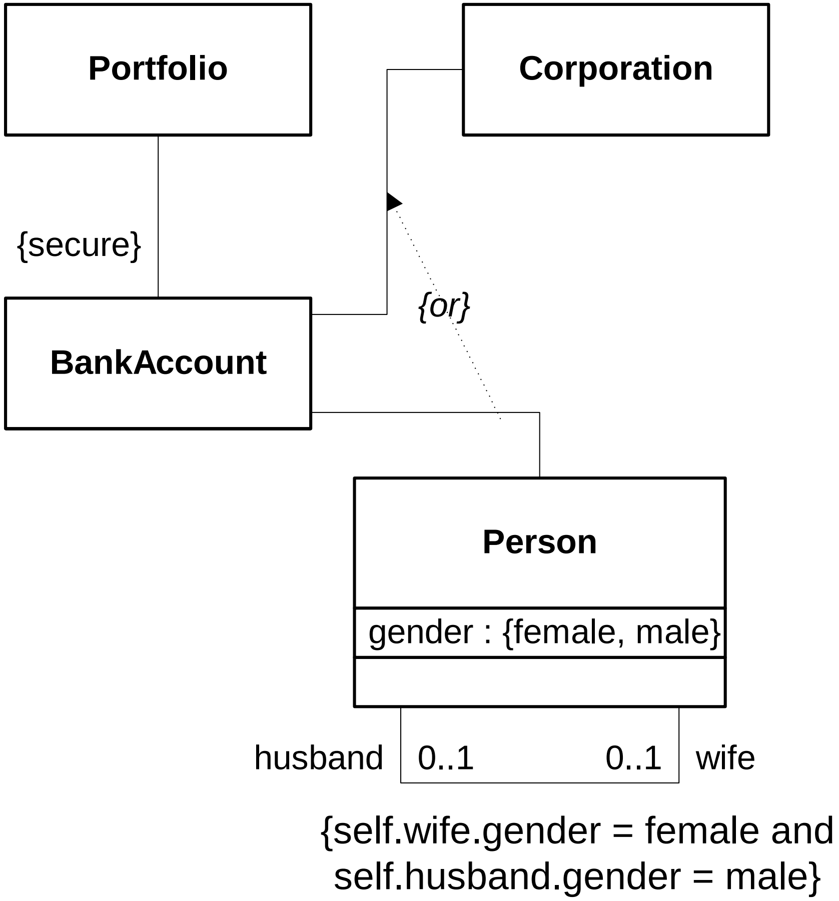
- “Realizes” AKA *implements*, *supports*, *matches*, etc.
- This means that class provides all the operations in the interface (and more?)
  - Remember, no implementation in interface definition
- Realization shown with dashed line, hollow arrow
  - Like dependency plus generalization
- Why have this?
  - Just factor out common functionality?
- Better “pluggability”, extensibility

Abstract Classes

- Implementation not provided for one or more operations
  - So, a subclass must extend this to provide implementations
- How to show this in UML?
  - Either italics for class name and operations
  - Or, use {abstract} property by name
- An abstract class with no attributes and all abstract operations is effectively an interface
  - But Java provides a direct implementation

Constraints

- Conditions that restrict values, relationships,…
- Can be free text or Object Constraint Langauge (OCL) (see textbook)
- Recommendation: Use sparingly!
- This example: from *UML User Guide*, p. 82



Identifying Classes for Requirements

- From textual descriptions or requirements or use cases, how do we get classes?
- Various techniques, and practice!
  - Key Domain Abstractions:
    - Real-world entities in your problem domain
  - Noun identification
    - Not often useful (but easy to describe)
- Remember: external view of the system for requirements
  - Not system internals, not design components!

Noun Extraction

- Take some concise statement of the requirements
- Underline nouns or noun phrases that represent things
  - These are *candidate classes*
- Object or not?
  - Inside our system scope?
  - An event, states, time-periods?
  - An attribute of another object?
  - Synonyms?
- Again, looking for “things”

Identifying Good Objects

- Don’t forget from earlier:
  - attributes and operations are encapsulated in objects
  - objects have a life-cycle
- Also, don’t worry about user interface
  - Think of user-commands as being encapsulated in the actors
- Consider:
  - Collections, things in a container
  - Roles
  - Organizations

Actors and Classes

- In some diagrams, actors represented as class boxes
  - With special stereotype above class name: <<actor>>
- UML allows special graphical symbol (e.g. a stick figure) to replace stereotyped classes
  - See Richter, p. 53

Exercise

Low Level Design using Sequence Diagrams

Sequence Diagrams

- X-axis is objects
  - Object that initiates interaction is left most
  - Object to the right are increasingly more subordinate
- Y-axis is time
  - Messages sent and received are ordered by time
- Object life lines represent the existence over a period of time
- Activation (double line) is the execution of the procedure.

UML sequence diagrams

- **sequence diagram**: an "interaction diagram" that models a single scenario executing in the system
  - perhaps 2nd most used UML diagram (behind class diagram)

- relation of UML diagrams to other exercises:
  - CRC cards -> class diagram
  - use cases -> sequence diagrams

Key parts of a sequence diag.

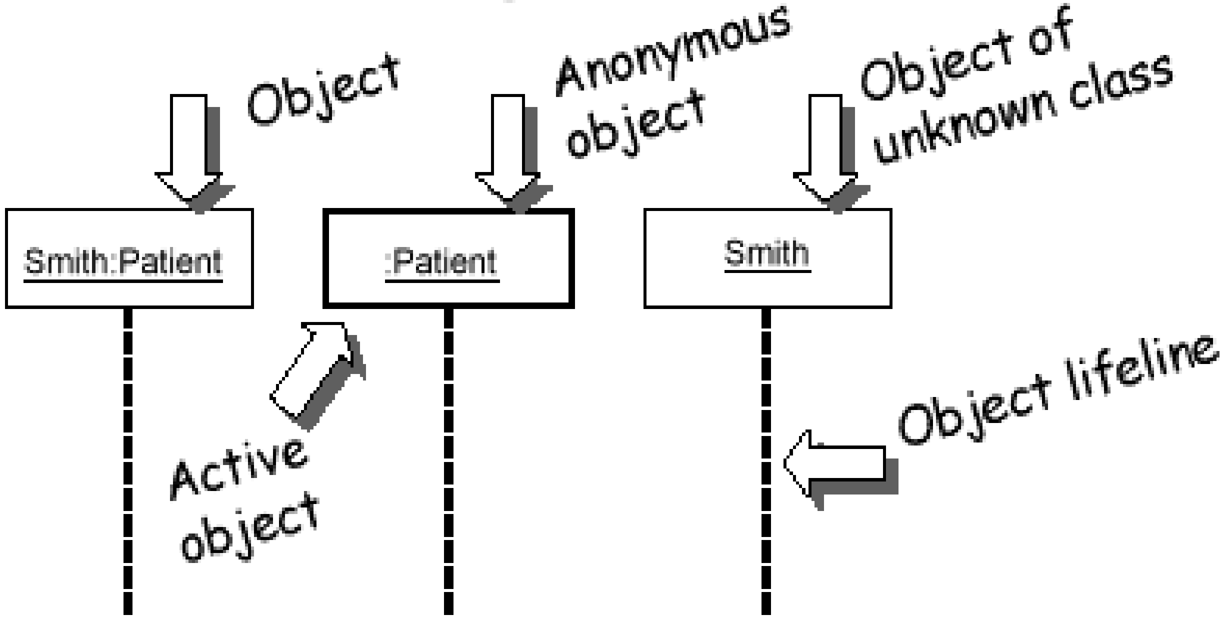
- **participant**: an object or entity that acts in the sequence diagram
  - sequence diagram starts with an unattached "found message" arrow

- **message**: communication between participant objects

- the axes in a sequence diagram:
  - horizontal: which object/participant is acting
  - vertical: time (down -> forward in time)

Representing objects

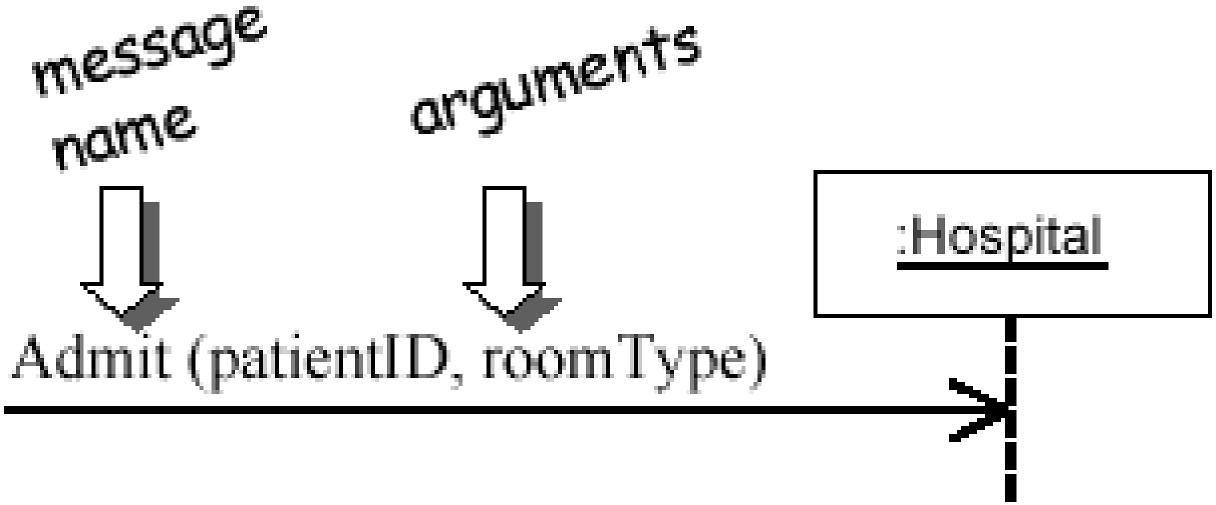
- Squares with object type, optionally preceded by object name and colon
  - write object's name if it clarifies the diagram
  - object's "life line" represented by dashed vert. line



Name syntax: <objectname>:<classname>

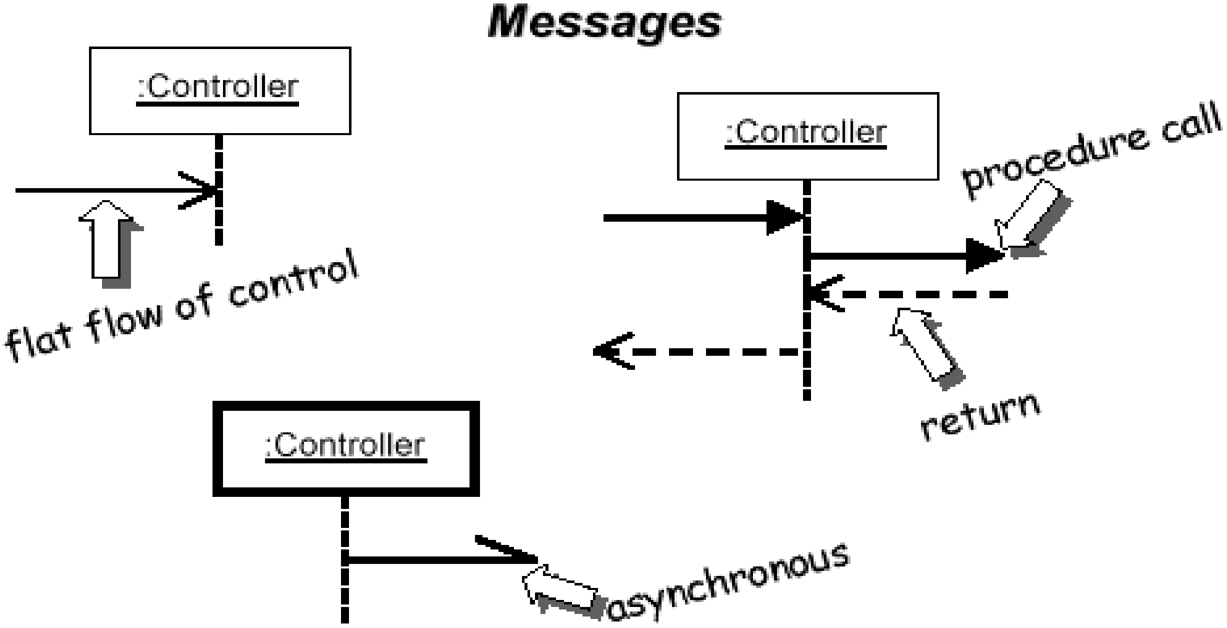
Messages between objects

- message (method call) indicated by horizontal arrow to other object
  - write message name and arguments above arrow



Messages, continued

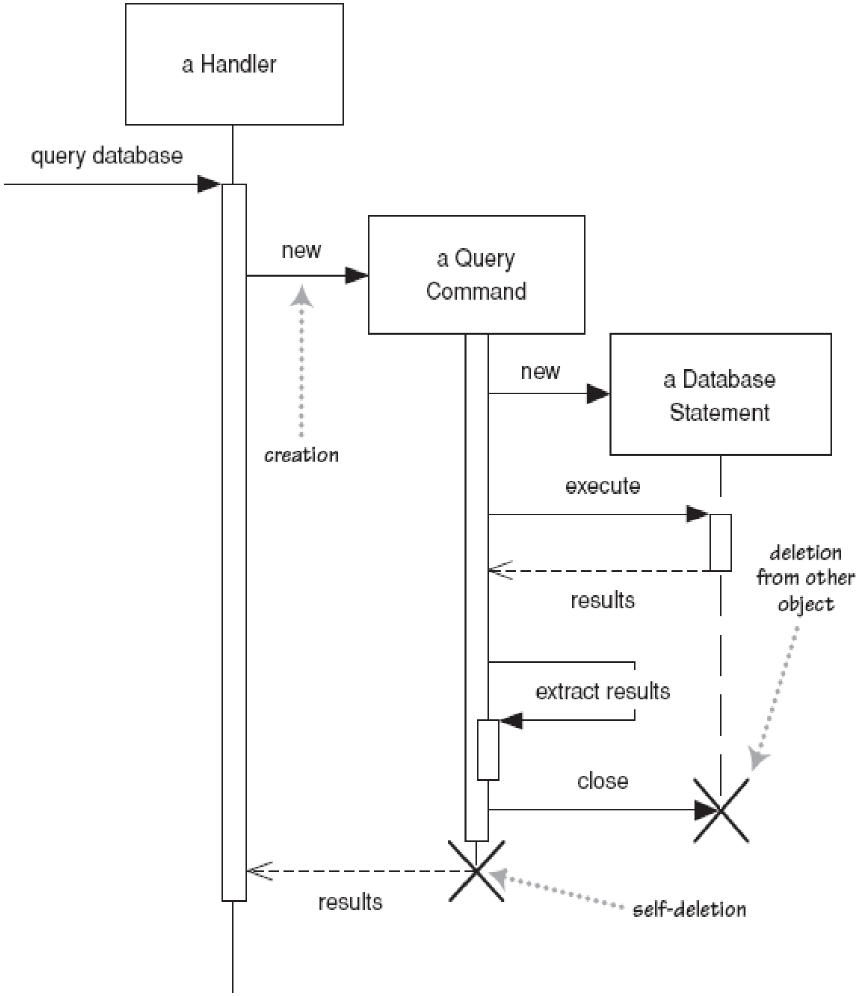
- message (method call) indicated by horizontal arrow to other object
  - dashed arrow back indicates return
  - different arrowheads for normal / concurrent



Lifetime of objects

- *creation*: arrow with 'new' written above it
  - notice that an object created after the start of the scenario appears lower than the others

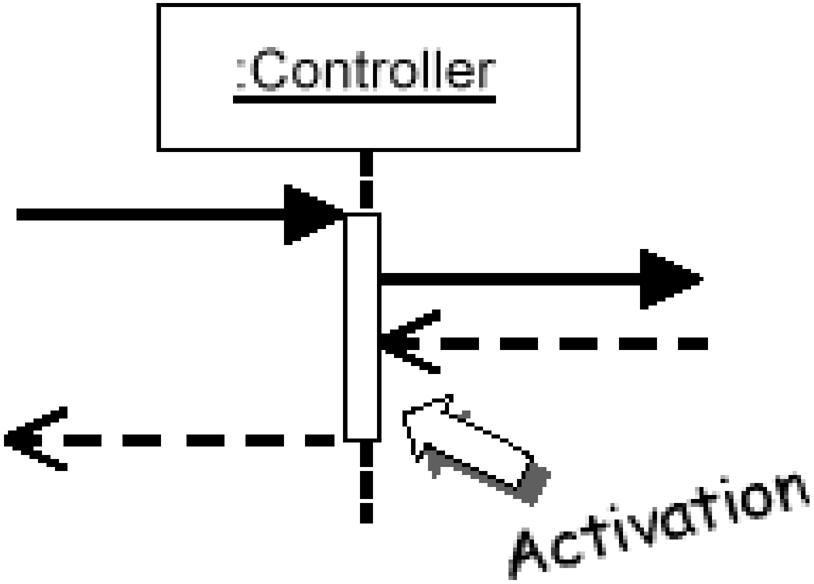
- *deletion*: an X at bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage-collected

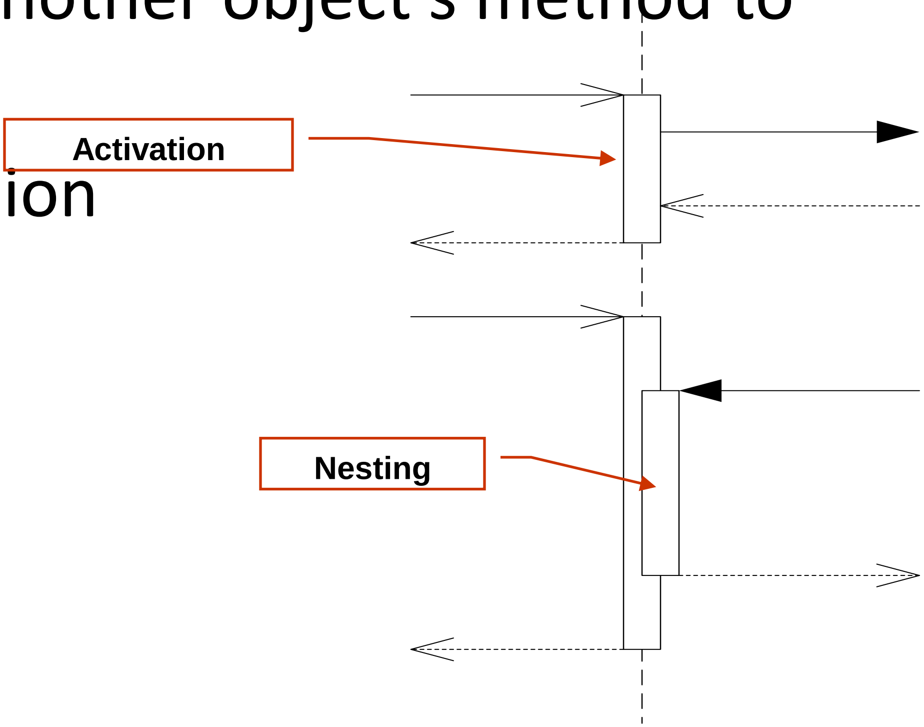


Indicating method calls

- **activation**: thick box over object's life line; drawn when object's method is on the stack
  - either that object is running its code, or it is on the stack waiting for another object's method to finish

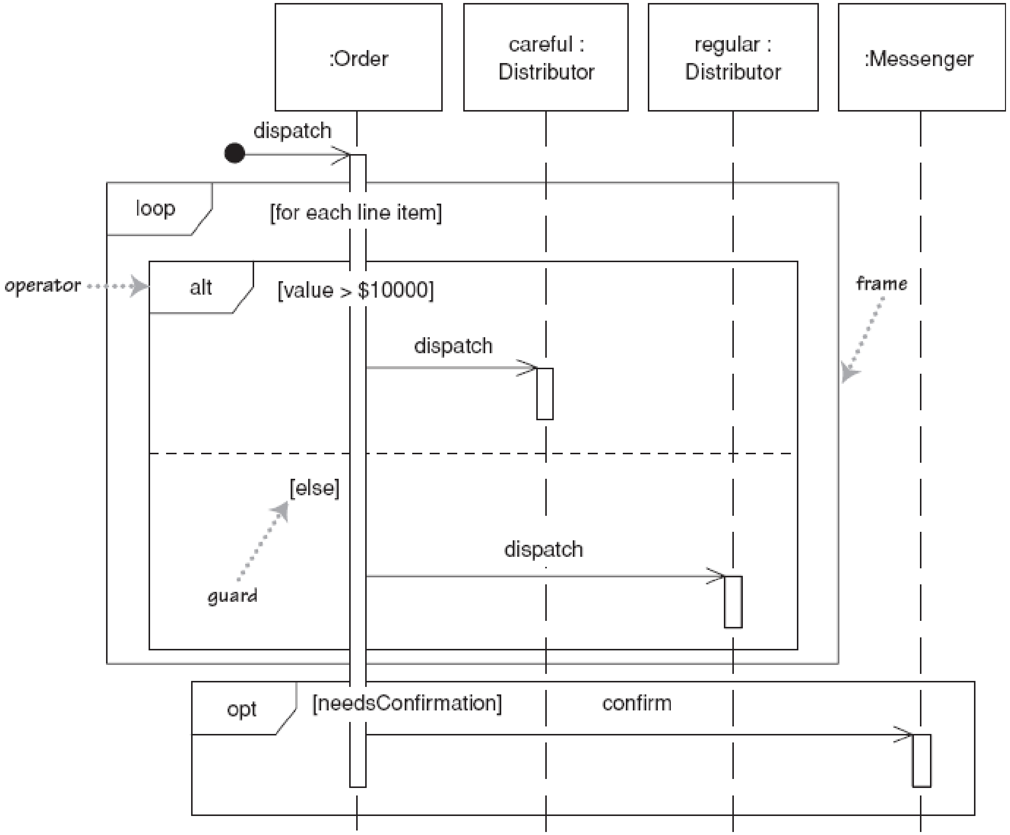
ırsion





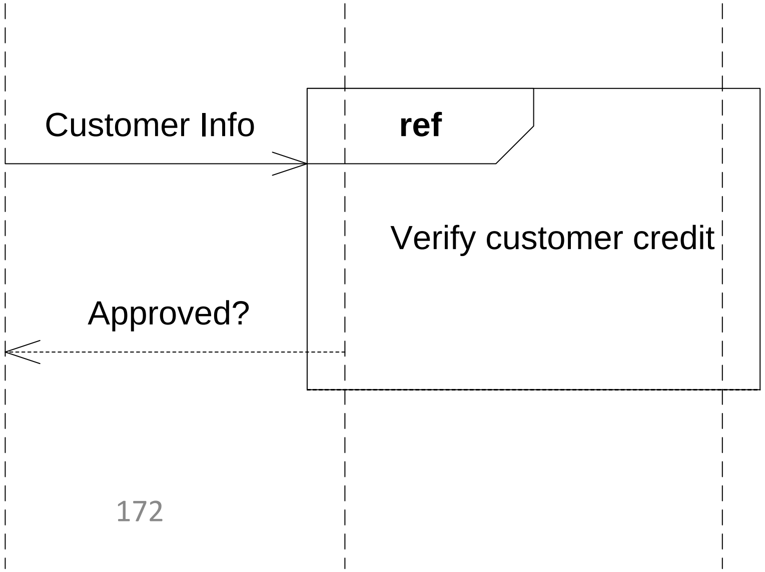
Indicating selection and loops

- frame: box around part of a sequence diagram to indicate selection or loop
  - `if` -> (opt) [condition]
  - `if/else` -> (alt) [condition], separated by horizontal dashed line
  - loop



linking sequence diagrams

- if one sequence diagram is too large or refers to another diagram, indicate it with either:
  - an unfinished arrow and comment
  - a "ref" frame that names the other diagram
  - when would this occur in our system?



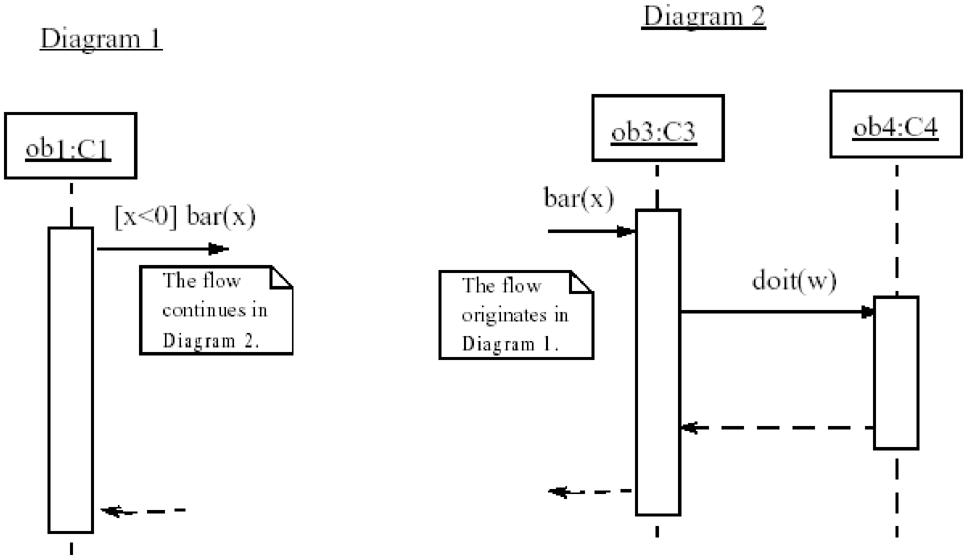
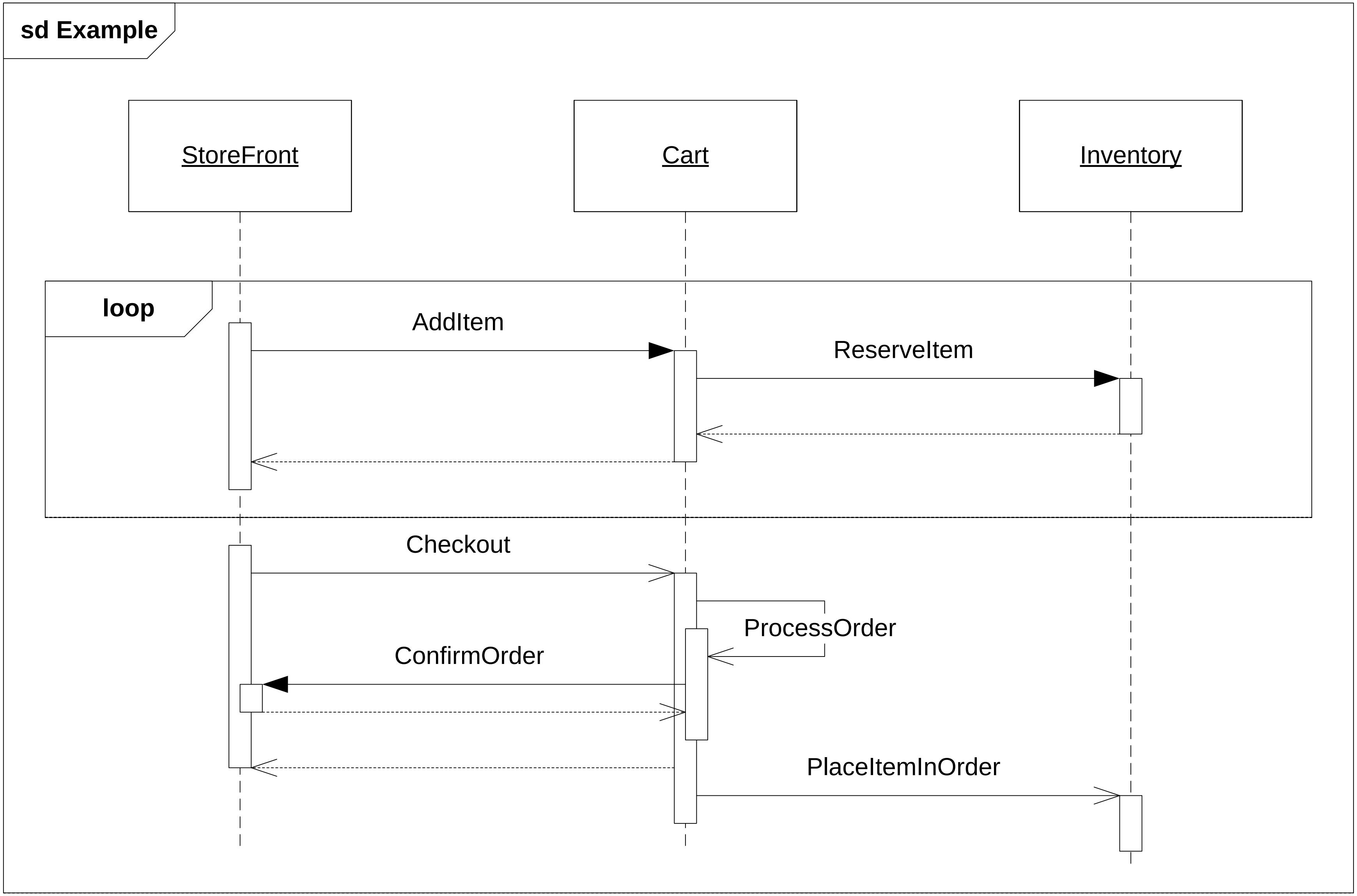


Diagram 1
ob1:C1
[x<0] bar(x)
The flow continues in Diagram 2.
Diagram 2
ob3:C3
ob4:C4
bar(x)
doit(w)
The flow originates in Diagram 1.

Example sequence diagram



Why not just code it?

- Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?
  - a good sequence diagram is still a bit above the level of the real code (not all code is drawn on diagram)
  - sequence diagrams are language-agnostic (can be implemented in many different languages
  - non-coders can do sequence diagrams
  - easier to do sequence diagrams as a team
  - can see many objects/classes at a time on same page (visual bandwidth)

Thank you

cpp.ajaypatil@gmail.com