Recitation 07

Inheritance – Error handling – Big-Oh

George Mappouras

10/28/2015

Inheritance 1

- Connect classes that you want to share variables and methods

```
class BankAccount{
…
};

class InvenstmentAccount: public BankAccount{
…
}
```

Inheritance 1

- Connect classes that you want to share variables and methods

```
class BankAccount{
…
};

class InvenstmentAccount: public BankAccount{
…
}
```

Inheritance 2

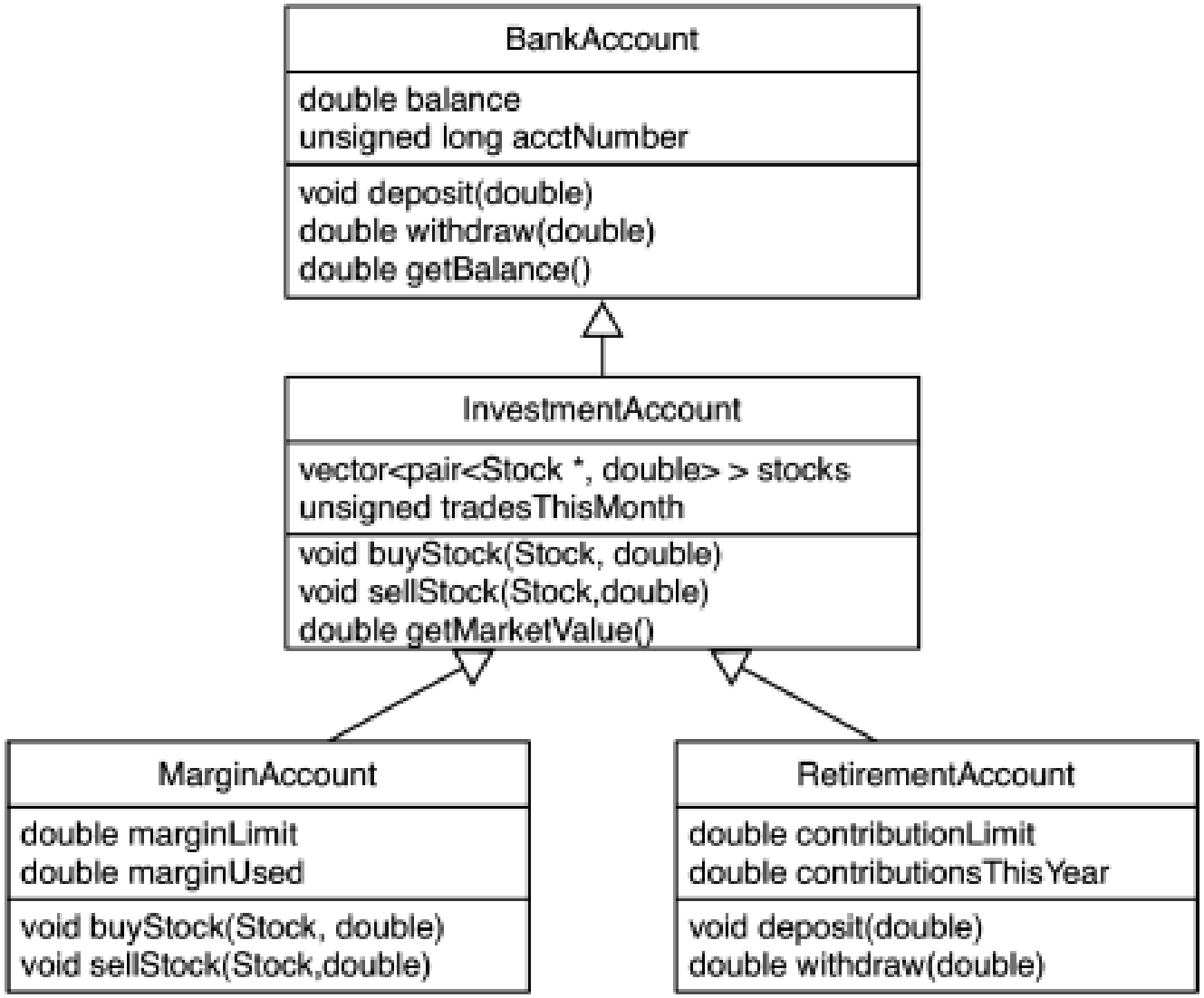
- BankAccount is the “parent” class and InvestmentAccount the “child”

```
class BankAccount{
…
};

class InvenstmentAccount: public BankAccount{
…
}
```

Inheritance 3

- Multiple levels of inheritance
- Each level may have multiple children



Inheritance and permissions

```
class BankAccount{
    private:
    int balance;
    protected:
    std::string name;
    public:
    long number;
};
```

```
class example1: public BankAccount{
…
}

class example2: protected BankAccount{
…
}

class example3: private BankAccount{
…
}
```

Inheritance and permissions

- None of the examples1-3 are aware of the private variable “balance”
- In example1 all inherited variables keep the same protection level as they had in the parent
- In example2 all inherited variables are protected and thus can only be accessed by example2 and its children
- In example3 none other than example3 can access the inherited variables

Is this legal? (1)

```
class BankAccount{
   private:
   int balance;
};
```

```
class CreditAccount: public BankAccount{
   private:
   int credit;
   public:
   void incBalance (int x){
     balance += (credit + x);
   }
}
```

Is this legal? (1) => NO

```
class BankAccount{
    private:
    int balance;
};
```

```
class CreditAccount: public BankAccount{
    private:
    int credit;
    public:
    void incBalance (int x){
      balance += (credit + x);
    }
}
```

Is this legal? (2)

```
class BankAccount{
    protected:
    int balance;
};
```

```
class CreditAccount: public BankAccount{
    private:
    int credit;
    public:
    void incBalance (int x){
      balance += (credit + x);
    }
}
```

Is this legal? (2) => YES

```
class BankAccount{
    protected:
    int balance;
};
```

```
class CreditAccount: public BankAccount{
    private:
    int credit;
    public:
    void incBalance (int x){
      balance += (credit + x);
    }
}
```

Is this legal? (3)

```
class BankAccount{
    public:
    int balance;
};
```

```
class CreditAccount: public BankAccount{
    private:
    int credit;
    public:
    void incBalance (int x){
      balance += (credit + x);
    }
}
```

Is this legal? (3) => YES

```
class BankAccount{
    public:
    int balance;
};
```

```
class CreditAccount: public BankAccount{
    private:
    int credit;
    public:
    void incBalance (int x){
      balance += (credit + x);
    }
}
```

```
class Animal
{
  protected:
     int numberOflegs;

  public:
     void setnumberOflegs(int n){numberOflegs = n;}
     int getnumberOflegs(){return numberOflegs;}
};
```

```
class Dog:public Animal{
     private:
        char *dogName;
     public:
        void setdogName(char *name){dogName = name;}
        char *getdogName(){return dogName;}
};
```

```
void main()
{
   Dog d;
   d.setnumberOflegs(4);//setnumberOflegs was
inherited from the Animal class
   d.setdogName("Aluk");
   cout<<"Dog Name:"<<d.getdogName()<<endl;
   cout<<"Number of legs: "<<d.getnumberOflegs()<<endl;
  //getnumberOflegs was inherited from the Animal class
}
```

Inheritance – Constructors and Destructors

- We Construct starting from the parent and moving to the children
- We Destruct starting from the children and moving to the parents

Method Overriding and Dynamic Dispatch

example from AOP

```
class A {

public:
  void sayHi() {
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

VS

```
class A {
public:
  virtual void sayHi() {    //note the "virtual" at the front
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  virtual void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

Method Overriding and Dynamic Dispatch

example from AOP

```
class A {

public:
  void sayHi() {
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

VS

```
class A {
public:
  virtual void sayHi() {    //note the "virtual" at the front
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  virtual void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

Method Overriding and Dynamic Despatch

What would the result be for the two different codes?

```
A anA;
B aB;
A * ptr = &aB;
anA.sayHi();
aB.sayHi();
ptr->sayHi();
```

Method Overriding and Dynamic Dispatch

example from AOP

```
class A {

public:
  void sayHi() {
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

Answer: A B A

VS

```
class A {
public:
  virtual void sayHi() {    //note the "virtual" at the front
    std::cout << "Hello from class A\n";
  }
};

class B : public A {
public:
  virtual void sayHi() {
    std::cout << "Hello from class B\n";
  }
};
```

Answer: A B B

Abstract Methods

```
class Shape {
public:
    virtual double getArea() const = 0;
    virtual ~Shape() {}
};
```

Lets write the Rectangle class

```
class Rectangle: _______ {
    double width;
    double height;
public:
    Rectangle(double w, double h): width(w), height(h) {}
    _________ getArea() const {
       return width * height;
    }
};
```

Lets write the Rectangle class

```
class Rectangle: public Shape {
    double width;
    double height;
public:
    Rectangle(double w, double h): width(w), height(h) {}
    virtual double getArea() const {
       return width * height;
   }
};
```

Question 18.3 : Given the following classes:

```
1  class A {
2    int x;
3    public:
4      void something() { ... }
5  };
6
7  class B : public A {
8   int y;
9   public:
10    void anotherFunction() { ... }
11 };
```

If you try to write this code:

```
1  A * ptr = new B();
2  ptr->anotherFunction();
```

You will receive a compiler error. Why?

How could we fix this problem by modifying class A and B?

Question 18.3 : Given the following classes:

```
1  class A {
2    int x;
3    public:
4      void something() { ... }
5  };
6
7  class B : public A {
8   int y;
9   public:
10    void anotherFunction() { ... }
11 };
```

If you try to write this code:

```
1  A * ptr = new B();
2  ptr->anotherFunction();
```

You will receive a compiler error. Why?

How could we fix this problem by modifying class A and B?

Question 18.7

```
#include <iostream>
#include <cstdlib>

class A {
protected:
  int x;
public:
  A(): x(0) {     std::cout <<"A()\n"; }
  A(int _x): x(_x) {  std::cout <<"A("<<x<<")\n"; }
  virtual ~A() { std::cout <<"~A()\n"; }
  int myNum() const { return x; }
  virtual void setNum(int n) {  x = n; }
};

class B : public A {
protected:
  int y;
public:
  B(): y(0) { std::cout <<"B()\n"; }
  B(int _x, int _y): A(_x), y(_y) {
    std::cout <<"B("<<x<<","<<y<<")\n";
  }
  virtual ~B() {  std::cout <<"~B()\n"; }
  virtual int myNum() const { return y; }
  virtual void setNum(int n) { y = n; }
};
```

```
int main(void) {
  B * b1 = new B();
  B * b2 = new B(3, 8);
  A * a1 = b1;
  A * a2 = b2;
  b1->setNum(99);
  a1->setNum(42);
  std::cout << "a1->myNum() = " << a1->myNum() << "\n";
  std::cout << "a2->myNum() = " << a2->myNum() << "\n";
  std::cout << "b1->myNum() = " << b1->myNum() << "\n";
  std::cout << "b2->myNum() = " << b2->myNum() << "\n";
  delete b1;
  delete a2;
  return EXIT_SUCCESS;
}
```

Answer to Question 18.7

```
A()
B()
A(3)
B(3,8)
a1->myNum() = 0
a2->myNum() = 3
b1->myNum() = 42
b2->myNum() = 8
~B()
~A()
~B()
~A()
```

Error Handling

```
try {
  //code that might throw
}
catch(std::exception &e) {
  //code to handle a generic exception
}
```

An example of exception handling

```
#include <iostream>
#include <exception>
using namespace std;
int main () {
try {
    int* myarray= new int[1000];
}
catch (std::exception& e) {
    cout << "Standard exception: " << e.what() << endl;
}
return 0;
}
```

Throw your own exceptions

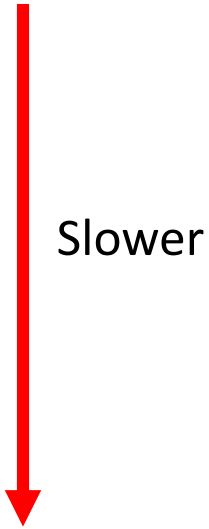
```
void withdraw_cash (double & balance, double amount){
if (balance < amount){
    throw InsufFunds;
}
balance -= amount;
}
```

```
double amount = 100;
double & balance = 50;
try{
    withdraw(balance, amount);
    amount = 50;
}
catch(InsufFunds & tmp){
    amount = 100;
}
```

Big – Oh Notation

We describe the worst case performance of a program with input size n

O(1) -> constant performance independent from the input
O(log(n)) -> logarithmic
O(n) -> linear
O(n*log(n)) Linearithmic
O(n^a) -> polynomial
O(a^n) -> exponential
O(N!) -> Factorial



Some Examples

O(1) ->

O(log(n)) ->

O(n) ->

Some Examples

O(1) -> Accessing an array element

O(log(n)) ->

O(n) ->

Some Examples

O(1) -> Accessing an array element

O(log(n)) -> Finding an element in a sorted array

O(n) ->

Some Examples

O(1) -> Accessing an array element

O(log(n)) -> Finding an element in a sorted array

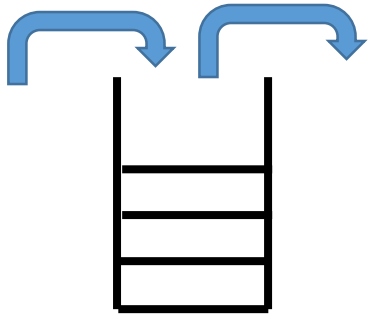
O(n) -> Finding an element in a non-sorted array

Common Structures

- Queues (FIFO)

Example: Waiting your turn in the DMV or Bank

- Stacks (LIFO)



Example: The stack of unwashed plates in your sink

- Maps

Example: Students and their uniqueIDs