Derived Classes

Eden Burton <ronald.burton@senecacollege.ca>
github repository:
(https://github.com/Seneca-OOP244/SCD-Notes)

The First Half in a Nutshell

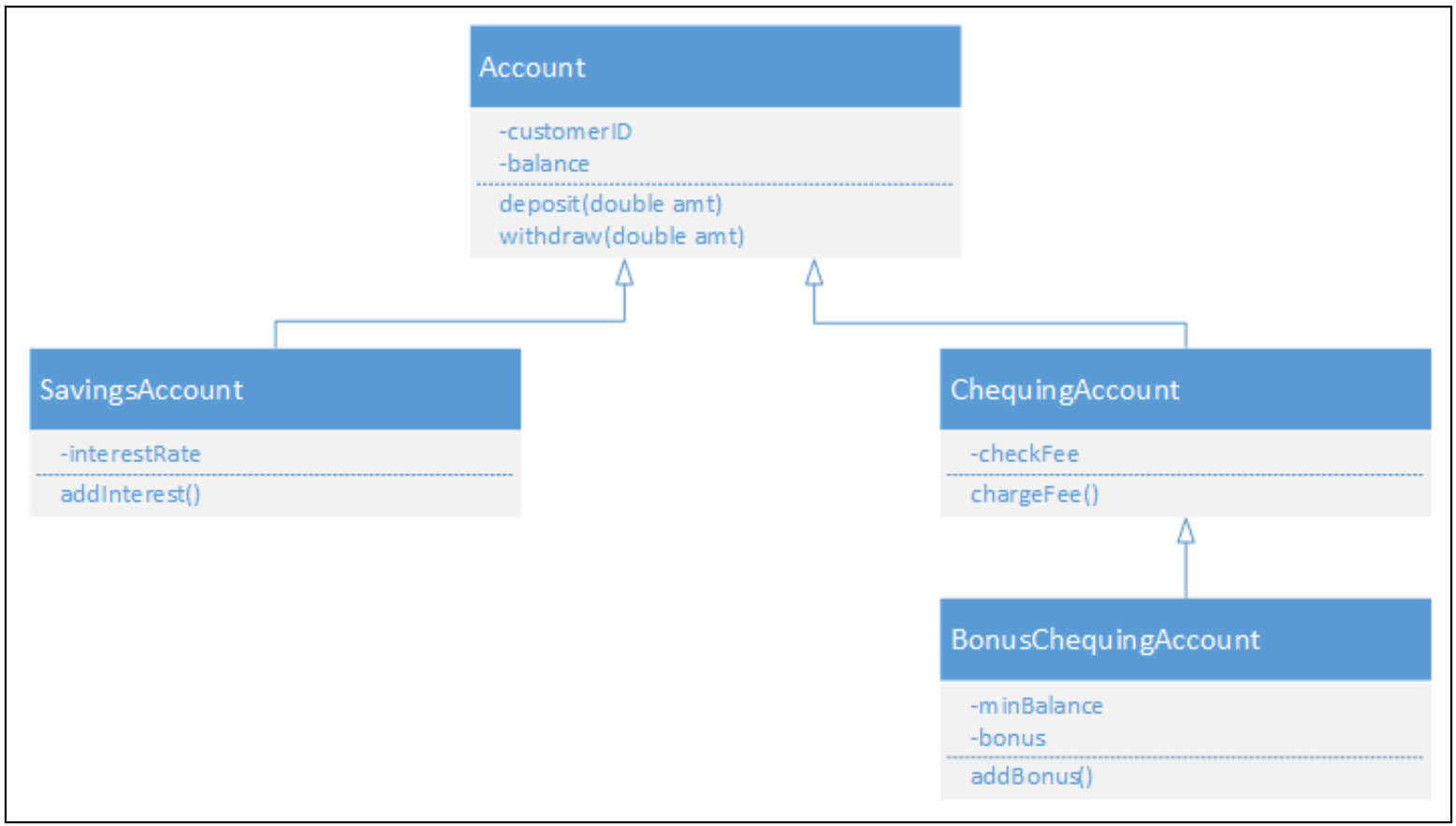
1. encapsulation
   - couples *data* and *logic*
2. classes describe objects
   - defines object type
   - describe show how object can be manipulated (methods)
   - states what entities can modify it (privacy levels)
3. constructors
   - used to properly initialize object, (optionally secure resources)
   - multiple initializations methods for different use cases
   - *destructors* ensure resources are freed

OK, Maybe Two Nutshells .... Design Goals

1. ensure that valid state is maintained throughout an object's lifetime
2. all methods retain valid state
3. user interaction with object is restricted through privacy levels
4. dynamic memory (resources) are managed correctly

Hierarchies

- type classification
- defines “is-a" relationships between types



Inheritance

".. to receive from an ancestor .."

- instance variables
- normal member functions

```
class Account {
    int customerID;
    double balance;
public:
    void deposit(double amt);
    void withdraw(double amt);
};

class SavingAccount : public Account {
    double interestRate;
public:
    void addInterest();
};
```

Inheritance Example

Member Functions in Derived Classes

- derived member function shadows base functions with same name

```
void Account::deposit(double amt) {
    balance += amt;
}

void SavingsAccount::deposit(double amt) {
    balance += amt;
    balance += 1;
}
...
int main() {
    SavingsAccount bobSavings;
    bobSavings.deposit(100.57);
}
```

Member Functions in Derived Classes

- use `Base::identifier` to access parent's version of method

```
void Account::deposit(double amt) {
    balance += amt;
}

void SavingsAccount::deposit(double amt) {
    Account::deposit(amt);
    balance += 1;
}
...
int main() {
    SavingsAccount bobSavings;
    bobSavings.deposit(100.57);
}
```

Derived Class Constructors

Construction Sequence

1. construct the base class portion of object
   - allocate memory for instance variables
   - execute the base class constructor
2. construct the derived class portion of object
   - allocate memory for instance variables
   - execute the base class constructor

Derived Class Constructor Arguments

- receive parameters for base class
- explictly use parameters for base constructor call

```
SavingsAccount::SavingsAccount
    (double bal, double rate)
     :  Account(bal)  {
     balance = bal;
...
int main() {
     SavingsAccount bobSavings(0,0.07);
}
```

Operator Overloading

- C++ allows refining the meaning of operators for compound types

```
struct Point {
    int x,y;
    Point& operator+=(const Point&);
};

bool Point::operator==
    (const Point& lSide, const Point& rSide) {
    return
    ((lSide.x==rSide.x) && (lSide.y==rSide.y));
}

Point& Point::operator+=(const Point& rSide) {
    x += rSide.x; y += rSide.y; return *this;
}
```

Operator Overloading

```
#include <iostream>
#include "Point.h"
using namespace std;

int main() {
    Point p1={2,3}; Point p2={4,5};
    Point p3 = p1 + p2;
    cout << "Equal: " << (p1 == p2) << endl;
    cout << "x: " << p3.x
         << " y: " << p3.y << endl;
}
```

Free Helpers

- does not need access to private members
- all information accessed through the public interface

```
class Point {
    int x,y;
 public:
    .....
    int getX() const {return x;};
    int getY() const {return y;};
};

areSame(const Point& lSide, const Point& rSide) {
    return  (lSide.getX() == rSide.getX())
            && (lSide.getY() == rSide.getY()) ;
}
```

Friends

- grants access to private members
- violate encapsulation rules

```
class Point {
    int x,y;
    friend bool operator==(const Point& lhs,
                           const Point& lhs );
    Point& operator+(const Point&);
};

bool Point::operator==(const Point& lSide,
                       const Point& rSide) {
    return ((lSide.x==rSide.x)
            && (lSide.y==rSide.y));
}
```