

# Review

Eden Burton <[ronald.burton@senecacollege.ca](mailto: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 how object can be manipulated (methods)
  - states what entities can modify it (privacy levels)
3. **objects** are runtime entities
  - can be created on the stack (static) or heap (dynamic)
  - statically created objects destroyed by program
  - dynamically created objects destroyed by programmer
4. **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

## Exam Notes

- structure concepts, coding, code interpretation
- workshops (esp 6-9)
  - 6 - overloading operators
  - 7 - derived classes
  - 8 - virtual functions
  - 9 - templates
- study all material but the focus is on the second half

# Key Topics

## 1. overloading operators

- member overloads
- friend (has access to private members)/free helper overloads

## 2. derived classes

## Key Topics II

### 1. polymorphism

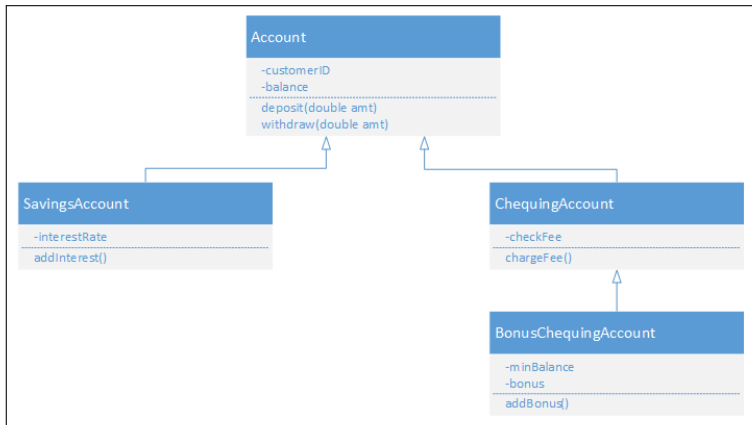
- coercion (arg type changed to allow access)
- function overloading (same identifier, different arg/return types)
- inclusion (member function selection based on type)
- parametric - same logic different types, compiler generates copy of implementations for each type.

### 2. abstract base class

- pure virtual function (add = 0); means class can be compiled but not instantiated
- derived class must implement pure virtual functions

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