KIE1008: Programming 2

Object-Oriented Programming: Composition

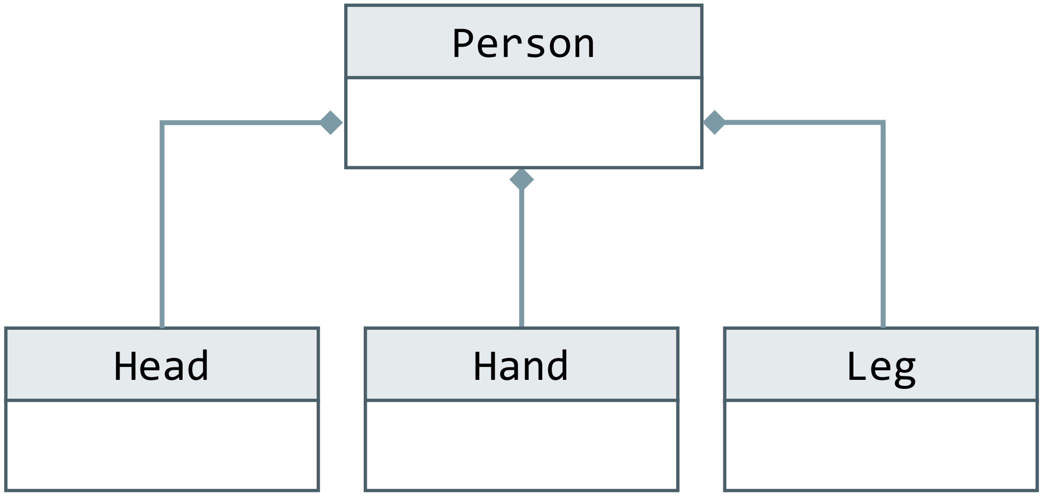
Semester 1, 2025/2026

Composition

- In real-world programming, the product or software comprises of many different smaller objects and classes. Since each of those objects performs a different task, they all are maintained in different classes.
- This process of building complex objects from simpler ones is called object composition. Object composition is used for objects that have a “has-a” relationship with each other.
- The most important advantage is if any changes have to be made in a child class, only the child class can be changed rather than changing the entire parent class.

Composition: Objects as Members of Classes

- Objects of one class as member data of other classes



```
class Time
{
  …
};

class AlarmClock
{
  …
private:
   Time t1;
};
```

Example 1: Composition (1/4)

```
class Point {
public:
    Point(int x = 0, int y = 0);
    int getX() const;
    int getY() const;
    void print() const;
private:
    int x, y;
};

class Location {
public:
    Location(Point, Point);
    double distance();
private:
    Point Source;  //composition
    Point Destination; //composition
};
```

Code: W03C01

Example 1: Composition (2/4)

```
Point::Point(int x, int y)
{
    this->x = x;
    this->y = y;
}

int Point::getX() const { return x; }
int Point::getY() const { return y; }

void Point::print() const
{
   cout << "(" << x << "," << y << ")";
}
```

Code: W03C01

Example 1: Composition (3/4)

```
Location::Location(Point s, Point d)
{
    Source = s;
    Destination = d;
}

double Location::distance()
{
    int x = Destination.getX() - Source.getX();
    int y = Destination.getY() - Source.getY();
    return sqrt(pow(x, 2) + pow(y, 2) * 1.0);
}
```

Code: W03C01

Example 1: Composition (4/4)

```
int main()
{
    Point p1(2, 7);
    Point p2(5, 9);
    Location loc(p1, p2);

    cout << "Distance between ";
    p1.print();
    cout << " and ";
    p2.print();
    cout <<  " is " << loc.distance() << endl;
    return 0;
}
```

Code: W03C01

Example 2: Composition (1/5)

```cpp
class Date
{
public:
    Date(int = 1, int = 1, int = 2000); // default constructor
    void print() const; // print date in month/day/year format
    ~Date();
private:
    int month;
    int day;
    int year;
};
```

Code: W03C02

Example 2: Composition (2/5)

```
Date::Date(int m, int d, int y)
{
   if (m > 0 && m <= 12) // validate the month
      month = m;
   else
      throw invalid_argument("month must be 1-12");

   year = y;
   day = d;
}


void Date::print() const
{
   cout << month << '/' << day << '/' << year;
}
```

Code: W03C02

Example 2: Composition (3/5)

```cpp
class Employee
{
public:
   Employee(const string &, const string &,
            const Date &, const Date &);
   void print() const;
   ~Employee();
private:
   string firstName;
   string lastName;
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
};
```

Code: W03C02

Example 2: Composition (4/5)

```
Employee::Employee(const string &first, const string &last,
   const Date &dob, const Date &doh )
   : firstName(first),
     lastName(last),
     birthDate(dob),
     hireDate(doh)
{ }


void Employee::print() const
{
   cout << lastName << ", " << firstName << "  Hired: ";
   hireDate.print();
   cout << "  Birthday: ";
   birthDate.print();
   cout << endl;
}
```

Code: W03C02

Example 2: Composition (5/5)

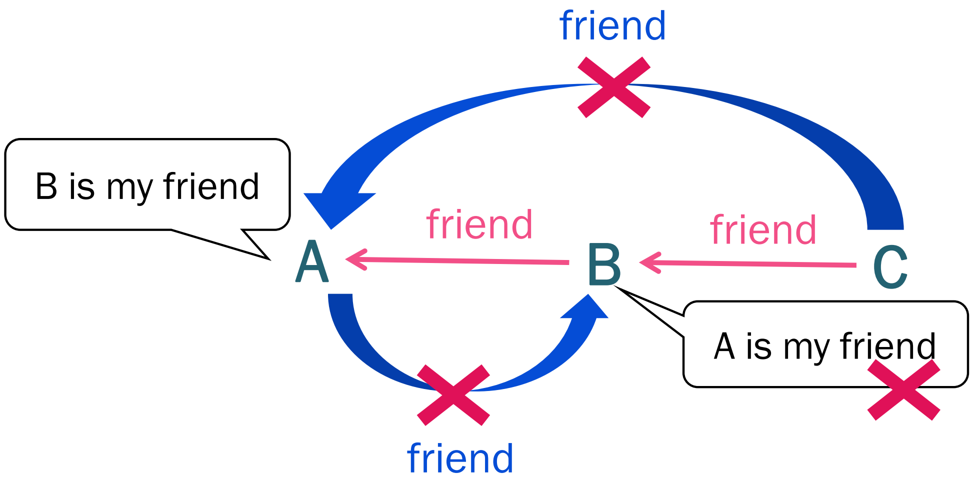
```
int main()
{
    Date birth(7, 24, 2000);
    Date hire(3, 12, 2020);
    Employee manager("Bob", "Blue", birth, hire);

    cout << endl;
    manager.print();
    return 0;
}
```

Code: W03C02

`friend` functions and `friend` classes

A `friend` function is a function defined outside the class, yet its argument of that class has unrestricted access to all the class members (private, protected and public).



1. Friendship is granted, not taken
2. Friendship is not symmetric
3. Friendship is not transitive

`friend` functions

- `friend` keyword should appear in the prototype only and not in the definition
- can be prototyped in either the private or public section of a class
- the object itself appears as an explicit parameter in the friend function (extra one parameter as compared to normal member function)
- need not use the scope resolution operator during definition

Example: `friend` function

```cpp
class Employee
{
  friend void setX(Employee &, int);
public:
   Employee() {count = 0;}
   int getCount() {return count;}

private:
  int count; // data member
};

void setX(Employee &c, int x)
{
  c.count = x; // allowed because setX is a friend
                  function of Employee class
}
```

```cpp
int main()
{
    Employee e1;
    cout << e1.getCount() << endl;

    setX(e1, 10);
    cout << e1.getCount() << endl;

    return 0;
}
```

Example: `friend` class

```
class Employee
{
   friend class Employer;
public:
   Employee(int c) {counter = c;}
private:
   int counter;
};

class Employer
{
public:
    Employer() {}
    int getCount (Employee &e)
       { return e.counter; }
};
```

```
int main()
{
    Employee e1(5);
    Employer m1;

    cout << m1.getCount(e1) << endl;

    return 0;
}
```

Code: W03C03

Proxy Classes

- Two of the fundamental principles of good software engineering are separating interface from implementation and hiding implementation details. However, headers do contain a portion of a class's implementation and hints about others.
- The use of proxy class allows private data to be hidden from clients of the class.

Example: Proxy Classes (1/4)

```
class Implementation
{
public:
    Implementation (int v) : value(v) {}
    void setValue(int v) {
        value = v;
    }
    int getValue () const {
        return value;
    }

private:
    int value; // data that we would like to hide from the client
};
```

Code: W03C04

Example: Proxy Classes (2/4)

```
class Implementation;  //forward class declaration

class Interface
{
public:
   Interface(int);
   ~Interface();
   void setValue (int);
   int getValue() const;

private:
   Implementation *ptr;
};
```

Identical public interface to that of class `Implementation`

The proxy class’s only private member is a pointer to an `Implementation` object

Code: W03C04

Example: Proxy Classes (3/4)

```cpp
Interface::Interface(int v) : ptr(new Implementation(v))
{
}

Interface::~Interface()
{
    delete ptr;
}

int Interface::getValue() const {
    return ptr->getValue();
}

void Interface::setValue(int v) {
    ptr->setValue(v);
}
```

The file Interface.cpp is provided to the client as a precompiled object code file. The client is not able to see the interactions between the proxy class and the proprietary class

Code: W03C04

Example: Proxy Classes (4/4)

```
int main()
{
   Interface i(5);

   cout << "(Before) Interface contains: " << i.getValue() <<
   endl;

   i.setValue(10);
   cout << "(After) Interface contains: " << i.getValue() <<
   endl;

   return 0;
}
```

Code: W03C04