School of Electrical Engineering and Computing

SENG2200/6220 Programming Languages & Paradigms

Topic 3

C++ vs Java – Part 2

Inheritance

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic 3 Overview

Composition and Aggregation
- Java References
- C++ Pointers, C++ Object Aggregation

Introduction to Inheritance
- Building and accessing objects under inheritance structures

Introduction to Polymorphism
- The role of Interfaces
- References, Pointers, and the IS A relationship

Inheritance Structures for Re-Use

Protected Attribute Data vs Public or Private

Composition vs Aggregation

Objects created (modelled) by joining other objects together

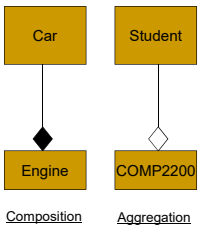
Eg a car is composed of: a body, an engine, four or five seats, a luggage compartment, and a fuel tank.

Eg a student is an aggregation of personal details, academic record, and currently enrolled courses.

The main difference is in the relative lifetimes of the component parts – composition implies an equality of lifetimes for the components, aggregation implies a possibility that component lifetimes do not match.

The level at which the object is modelled is important in deciding which of the component descriptions is used.

In UML



Composition Comparisons

Group
- Musician singer - Musician leadguitar - Musician bassguitar - Musician drummer - Person manager
<methods go here>

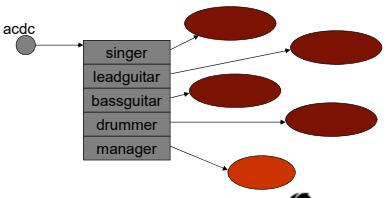
Java
- Components are references
- Need to be separately instantiated
- Relies on the garbage collector to recognize when each of the composite parts needs to be deallocated

C++
- Components are objects
- Instantiated and initialized using the default constructor
- When the outer object is destroyed the composite parts are deallocated as well

Composition - Java

Group
- Musician singer - Musician leadguitar - Musician bassguitar - Musician drummer - Person manager
<methods go here>

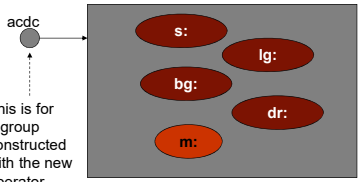
Java
- Components are references
- Need to be separately instantiated
- Write Java to create what is below
- How would aggregation be different?
  - We will find out in a couple of slides time



Composition – C++

Group
- Musician singer - Musician leadguitar - Musician bassguitar - Musician drummer - Person manager
<methods go here>

C++
- Components are objects
- Instantiated and initialized using the default constructor (or could receive parameters)
- Write C++ to create what is below
- How would aggregation be different? (next)



This is for a group constructed with the new operator

C++ Aggregation

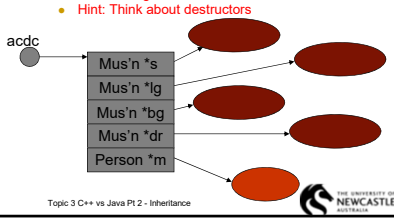
C++

- Components are pointers
- Need to be separately instantiated
- Write an explicit destructor
- Write C++ to create what is below
- How would composition be different?
  - This setup can be used
  - What changes?
  - Hint: Think about destructors

Group

- Musician singer
- Musician leadguitar
- Musician bassguitar
- Musician drummer
- Person manager

<methods go here>



Aggregation – Java (Notice anything special)

From Java Composition Slide (slide 5)

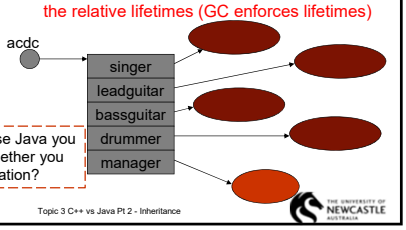
- How would aggregation be different?
- Components are references
- Need to be separately instantiated
- Write Java to create what is below
- It is exactly the same as the layout for composition – the programmer decides on the relative lifetimes (GC enforces lifetimes)

Group

- Musician singer
- Musician leadguitar
- Musician bassguitar
- Musician drummer
- Person manager

<methods go here>

Does this mean that if you use Java you don't have to worry about whether you have Composition or Aggregation?



Introduction to Inheritance

A major way of gaining *software re-use*

Allows a class to be developed directly from another class *without altering or copying the original*

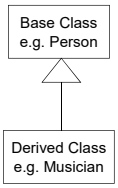
Basic relationship between the classes is the "IS A" relationship

E.G. A Musician IS A Person

when a musician object is created:

- A musician IS A person, so any musician object will have all the attributes of a person object
- The person-style methods also remain.
- Extra musician-style attributes are added in.
- Extra musician-style methods are also added.
- Scoping rules allow for person-style methods to have a new musician-style implementation added if required.

Inheritance In UML



Pointers and References under Inheritance

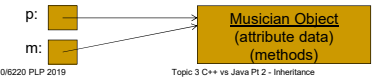
A Musician IS A Person, and therefore reference/pointer of type Person is able to refer to a Musician object.

Allowing a derived class object to be referred to as a base class object is integral to implementing the IS A relationship and so is basic to understanding inheritance.

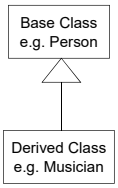
It is also the first step in allowing polymorphism.

E.g. A Musician Object IS A Person, so:

```
Person p;      // in Java -> in C++, Person *p
Musician m = new Musician(… constr params …);
p = m;         // This assignment is legal
```



Inheritance In UML



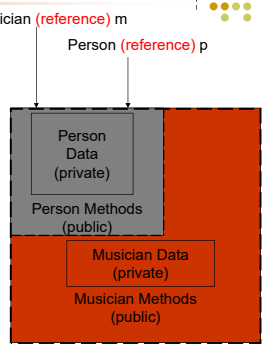
Anatomy of an object under inheritance

Person data remains private within the Person class and so is only accessible to the Musician methods via the Person methods

Musician data is private.

Scoping still allows methods to be redefined.

This is a very low level schematic view – normally you don't need to think about derived objects at this level.



Attributes and Methods for an Object under Inheritance

What is available? (What can be seen?)

Outside the Derived (Musician) class:

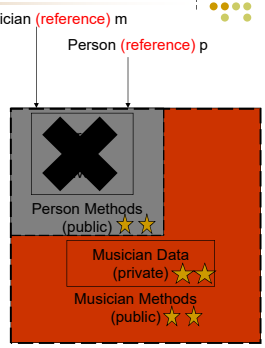
- Public methods of both the Base and Derived Classes.
- We can ask for BOTH Musician operations and Person operations.

Inside the Derived (Musician) class:

- Base class public methods
- Anything that is public or private in the derived class (i.e. ANYTHING)

What about the Base class private data?

- Only valid operations can be done (information hiding) and so the Base class public methods must be used to alter or access the Base class attribute data.



Adding and Redefining Attributes and Methods


```
public class Person  {
    private String name;
    private String addr;
    ……
    public String getName() {…};
    public String getAddr() {…};
    ……
}
```

A Student object, e.g.

```
s.getAddr();
```

will choose its latest, that is the Student, definition of getAddr()

```
public class Student extends Person {
    private int studentNo;
    private String termAddr;
    ……
    public int getStudNo() {…};
    public String getTermAddr() {…};
    public String getAddr()  {
        if UNI.isTermTime()
            return termAddr;
        else
            return super.getAddr();
    }
```


Inheritance in Java and C++

Inheritance in Java is simpler and more straight forward than in C++.

Java classes may extend a single class and implement as many interfaces as you like.

C++ classes may inherit from any number of classes, so multiple inheritance is allowed.

C++ allows the implementation of public, protected, and private inheritance (but only public inheritance properly implements the IS A relationship between the classes).

Java inheritance can be viewed as a refinement of C++ inheritance structures, making useful features easier to use, and leaving less useful features out of the language.

Building an object under Inheritance

The object is built from the inside first.

class B extends A in Java (or class B : public A in C++) means that

- The class A part of the object is built first by way of the class A constructor
- Then the class B extensions are added by way of the class B constructor
- In C++ this may involve aggregated sections being implicitly instantiated and initialized by their default constructor
- In Java, any first class objects aggregated within the object must be explicitly instantiated. This is also the case for C++ if explicit pointers are used (whereupon an explicit destructor is usually required).

Chaining constructor calls

Java: There can be only one parent class called the super class

Java is therefore able to use the keyword call super to explicitly call the constructor of its parent class

Parameters may be passed via the call as in super(… , … , …);

If no reference is made to super then an implicit call is made to super( ); (default constructor).

C++: The possibility of multiple inheritance in C++ means that explicit calls to a parent class constructor must explicitly give the name of the constructor (the particular class),

e.g. Person(… , …);

Under multiple inheritance base class sections of the object can be made in a specific order.

Inheritance – When and Where?

We know HOW to implement inheritance in Java and C++

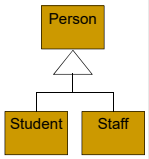
When and where do we use it?

- This is a major part of O-O Analysis and Design

Mostly we do not think about having a Person object and then deriving a Student object from it.

A Uni-based problem will have a whole lot of different people that our problem statement will initially talk about as Student(s) and Staff (as a simple example).

- We set up specs (a class) for a set of Students
- We set up specs (a class) for a set of Staff members
- THEN, we realise that there are attributes and methods in common
- We extract (factor out) these common specs into a base class and decide that Person is a good name for it.
- Later we might decide that a Contractor can also inherit from Person



University Class Hierarchy

The Java Class called Object

Unlike C++, Java is a pure O-O language

- Remember that C++ is C with O-O extensions

A benefit Java has from this is that it defines a standard base class called Object from which all other classes, including programmer defined classes are derived.

If you write a class in Java, and do not specify any inheritance-style derivation for it, then it is implicitly taken to be derived from class Object.

This allows certain standard features, required by all objects (no matter what their class) to be defined in their most basic form.

Object is an abstract class (see later)

Introduction to Polymorphism

19

Greek derivation – poly (many) morphic (shapes)

In language implementation terms, also referred to as late binding

O-O programming views objects as animate (alive), with the programmer sending a message to an object and the object responding – this implies that the object itself decides how it will respond to a particular message

It is possible to know what sort of behaviour (methods) a particular class will need, but to not know exactly how those methods will achieve their result(s) until more is known about the particular object.

E.G. A shape - We will know that we will have to calculate its area, but we will not know how to do so until we know exactly what sort of shape it is

Abstract Behaviour and Abstract Classes

20

Knowing that a particular operation (method) is needed but not knowing exactly how to perform it, results in the abstract specification of that method – you will know how to call the method, and what answer it will give you, you just won't know how it is to be done.

Any class that contains one (or more) method (s) of this type, is therefore an abstract class – it can be properly specified, but it cannot be properly implemented.

Such abstract classes therefore only exist for the purpose of having other more specialized classes (perhaps non-abstract classes) derived from them.

Once you have implementations of all methods you have a concrete class that can be instantiated – i.e. you can have actual objects of that class.

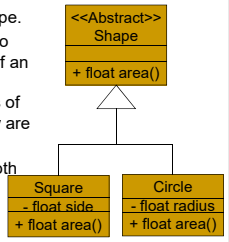
Abstract Classes & References

21

The special thing about a Shape (Slide 19) is that a shape may not know how to calculate its area until it knows that it is a circle or a square or a triangle, but it can still be looked at (or referred to) in terms of it being a basic shape.

Consequently, even though it is not possible to have an object instantiated as a member of an abstract class, once a concrete object has been created, it can be referred to in terms of any of its derivation ancestors, even if they are abstract classes.

So a Square object and a Circle object can both be referred to as Shape(s), and this is the key to making polymorphism work in practice.

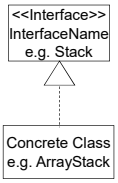


Interfaces

22

A class which contains nothing other than a set of abstract method specifications and no implementations of any of them, and no associated attribute data, is a special type of abstract class called an Interface.

The special thing about interfaces is that they (unlike a base class) do not bring any implementation at all to the concrete class (neither attributes, nor actual methods) and so they play a different role in the inheritance structures outlined earlier



UML diagram for Implements an Interface (dotted line)

Inheritance in Java

23

The basic mechanism for inheritance in Java is the keyword extends and this provides basic public inheritance.

- It is only possible to extend a single class
- Public items (mainly methods) remain public in the derived class
- Private items (data attributes and supporting functions) become inaccessible within the derived class
- Protected items remain protected in the derived class (there is an example of protected data on slides 29-32)

The Java keyword implements allows any number of interfaces to be added into a derived class as interfaces only add specification, not implementation.

- All methods of the interface MUST be implemented

Inheritance in C++

24

class B : public A ; // Public inheritance is the only IS A construct

- Public items remain public
- Protected items remain protected
- Private items become inaccessible
  - except via the public methods

class H : protected G ; // this is not part of SENG2200

- Public items become protected
- Protected items remain protected
- Private items become inaccessible

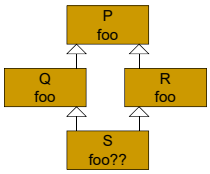
class K : private J ; // Often referred to as Implementation Inheritance

- Public and protected items become private
- Private items become inaccessible
- Can be used in place of composition

Multiple Inheritance in C++

class S : public Q, R { , … } ;

- Problems arise when both Q and R are derived from a common class (say class P)
- Do you have two copies of the attribute data?
- This is the so-called diamond inheritance problem



Interfaces in C++

There is no specific interface structure in C++, but …

The same thing can be obtained by using pure virtual classes (this is basically what an interface is), coupled with multiple (public) inheritance

The diamond inheritance problem does not exist as the pure virtual classes only add specification, not implementation.

Extra flexibility is possible as a class, so derived, can remain abstract until a derivation of it finally has all method specifications implemented.

Protected Data

This is attribute data in the base (ie parent) class that is labelled as protected rather than private.

The UML designation for protected data is #.

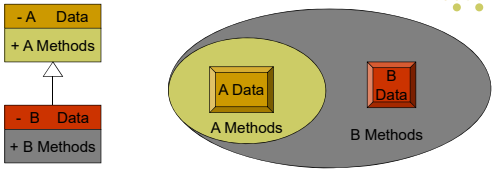
Protected data can be directly accessed within its own class in the same way that private data can.

However, it can ALSO be directly accessed (ie without using base class methods) from within any class that inherits from this class.

It is a breach of Information Hiding principles – however it does have a place in O-O design.

It is NOT simply a shortcut that allows a bad designer or bad programmer to be lazy.

Inheritance Hierarchies and Finding Item Names



Calling a public method - Start at the derived class and search up the inheritance hierarchy for a method implementation.

Within derived class – directly accessing data – Look at derived class private data, then look up the inheritance hierarchy looking for a protected data item in one of the super classes.

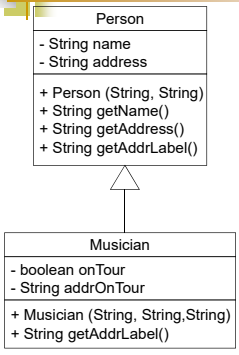
Protected Data vs Private Data

This is a breach of Information Hiding principles to provide (shortcut) direct access to data that would otherwise be private data in the parent class.

It is providing a shortcut to the programmer to remove the need for get and set methods that then need to be called every time the sub-class wants to fetch or alter the data item(s) in the super-class.

There are valid uses for it and there are unwise uses for it, as the following example shows.

Musician(s) and Person(s) - 1

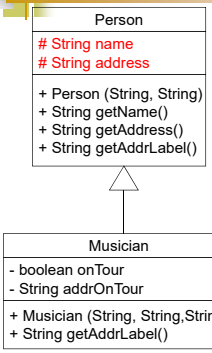


Java – Musician extends Person
C++ - Musician : public Person
Private data in Person becomes inaccessible in Musician

getAddrLabel method for Person
```
return name + address
```

getAddrLabel for Musician
```
if (onTour)
    return getName() + addrOnTour
else
    return Person::getAddrLabel()
    // or getName() + getAddress()
```

Musician(s) and Person(s) - 2



Java: Musician extends Person

C++: Musician : public Person

Protected Data in Person class remains visible in Musician

```
getAddrLabel method for Person
    return name + address

getAddrLabel for Musician
    if (onTour)
        return name + addrOnTour
    else
        return name + address
        // or Person::getAddrLabel()
        // as before
```


Protected Data

The breach of Information Hiding principles exists only across the "inheritance boundary" – that is from a super-class to its sub-class.

In O-O Analysis many super classes are discovered after their sub-classes by way of factorization (shown in slide 17).

This means that the super class only exists for the purpose of being implemented once and "derived from" many (>1) times.

So there is a high likelihood that the super class will never be instantiated in its own right.

As such, listing data as protected rather than private is often not as drastic a breach of information hiding as it may seem at first

But slide 5 shows this example is not such a good one

- What would make it OK?
- Hint: Is it correct to model a manager as a Person?