School of Electrical Engineering and Computing

SENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS

(S1, 2020)

Polymorphism

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

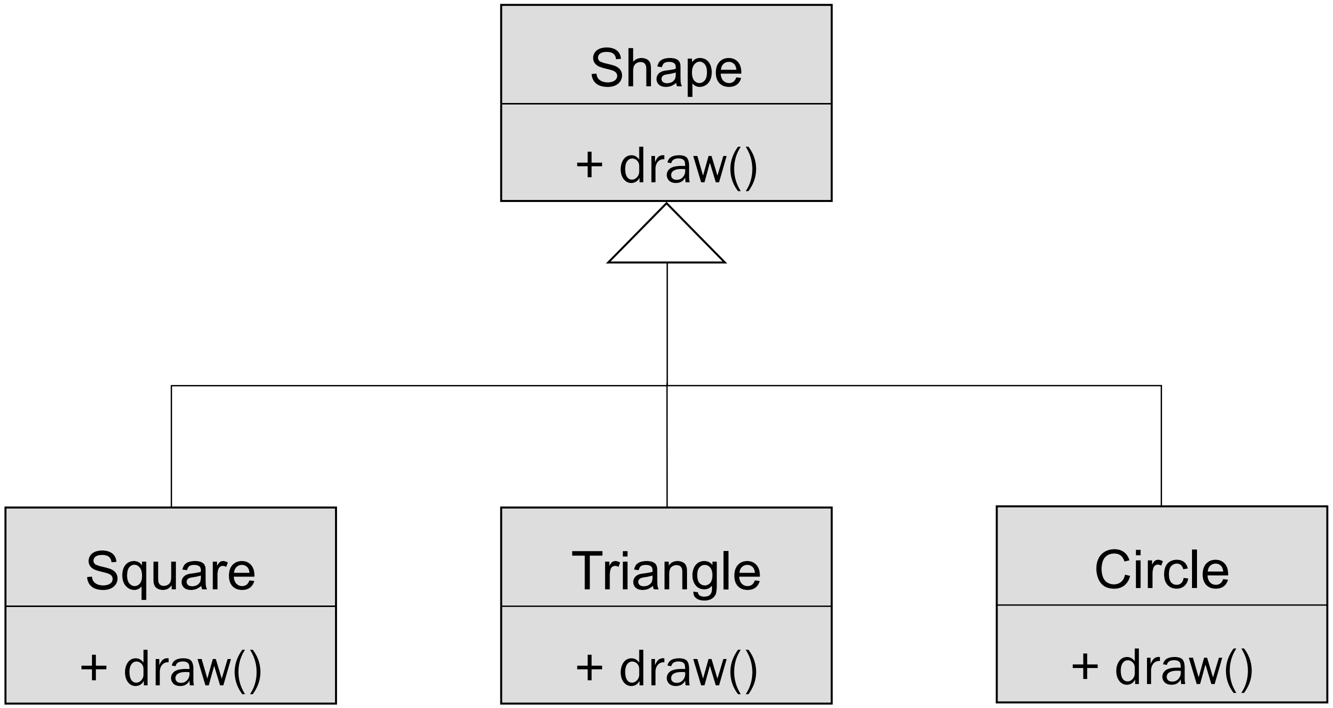
Outline

- The 4 attributes of true O-O Programming
  - *E(ncapsulation)*
  - *I(nformation)H(iding)*
  - *I(nheritance)*
  - *POLYMORPHISM*
- Polymorphism and Abstract Classes
- Interfaces revisited
- Aspect-Oriented Programming

The Attributes of O-O Programming

- ❖ Encapsulation
  - Placing related Data (attributes) and their related Operations (methods) together (e.g. C++ structs).
- ❖ Information Hiding
  - Making these related data attributes private to the "outside world" and only accessible by way of its own public interface, a set of operations (methods) which are the only way that a client can request changes to the attribute data (eg. C++, Java, C#, etc. classes) .
- ❖ Inheritance
  - Allowing automatic building of one set of objects (class) from another, without the need to modify the original (class) definition.
- ❖ POLYMORPHISM
  - Allowing an object to respond to a method call according to its own (class's) definition of what this method call should produce, irrespective of how that object is known to the program. Method calls are implemented as strict message passing.

Inheritance Hierarchy



Polymorphism

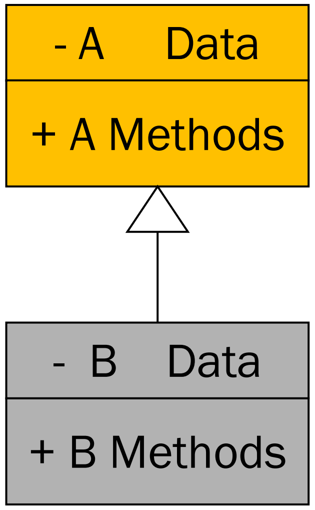
- Allows the same code to be used with different types of objects and behave differently.
  - *E.g., Shape object can behave as any of square, triangle or circle object.*
- Instead of invoking the methods of an object, we are actually doing is sending a message to the object.
  - *Consequently, the object should respond to the message according to its own definition, and this should not be influenced at all by the means by which the object is known to the program (how it is referenced).*

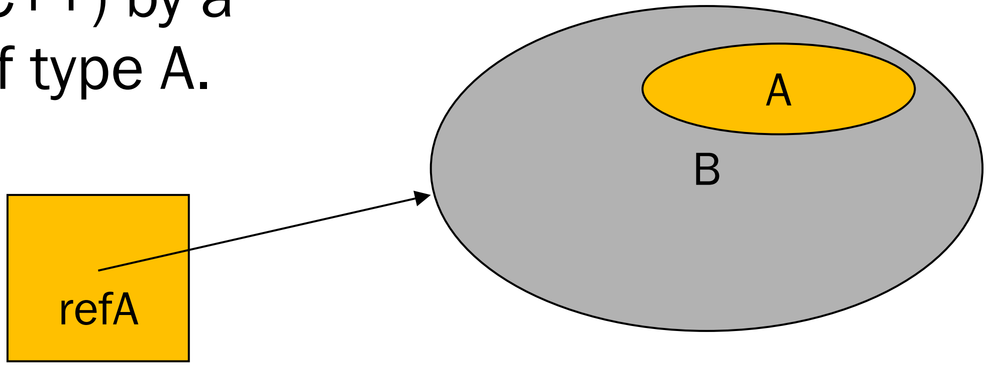
Polymorphism

- A message is of the form method_name(params) which is sent to the object, and the object then returns a response which is the return value for the method. The return value can be simple or complicated, (e.g., Java)
  - *void (no response),*
  - *a primitive value, or*
  - *a reference (to an object).*
- The effect of the method can be by way of
  - *changes to the object's attribute data,*
  - *changes to the attribute data of any of the reference parameter objects,*
  - *but also may include*
    - output to an output stream (or input from an input stream),
    - throwing of an exception.

The Key to Polymorphism

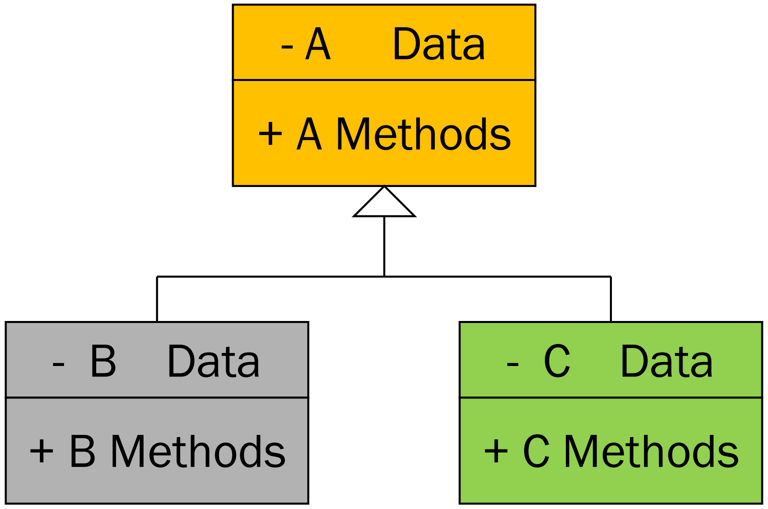
- The IS-A relationship of inheritance
- B is a derived class of A
  - *public class B extends A   // in Java*
  - *public class B : public A   // in C++*
- Because an object of type B is an object of type A, it can be referred to (Java), or pointed at (C++) by a reference or pointer of type A.





The Key to Polymorphism

- If we extend this to having classes B and C both inheriting from class A:
- A reference of type A (say refA) can refer to either a B object or a C object, as well as a type A object.
  - *An array of type A can therefore even contain objects of type B, or type C, because a reference of type A can refer to a B or C object as well as an A object.*
  - *The actual type of the object that refA refers to, or is referred to by any element of our array object, can only be determined by looking at the actual object as the program is being executed, ie. at run-time.*

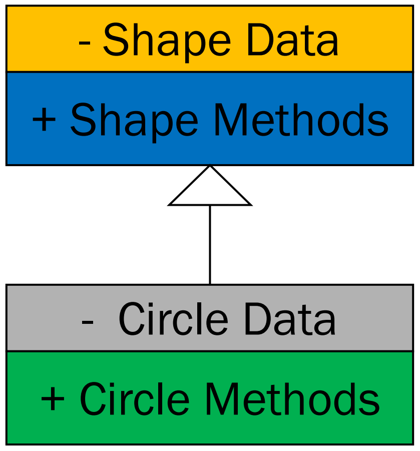


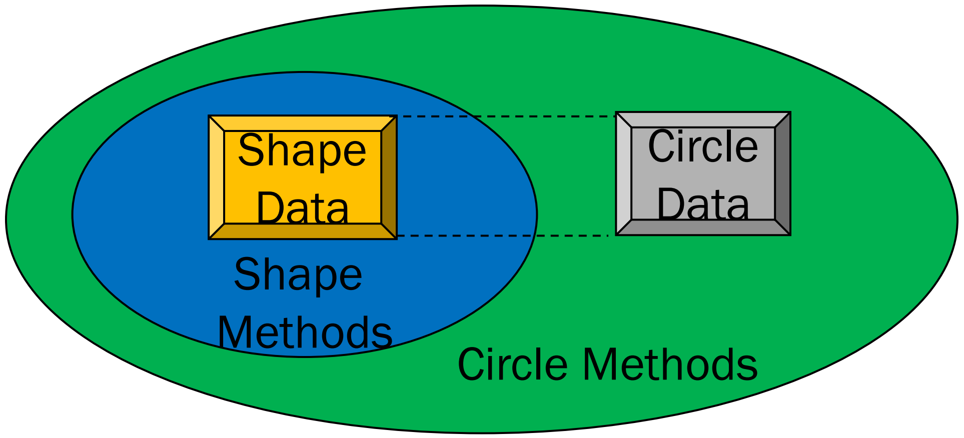
Exercise: Draw an array of type A in Java and C++.

Polymorphism and Late Binding

- Polymorphism
  - *Compile-time*
  - *Run-time*
- Polymorphism allows an object to respond to a message according to its own definition of what should be done.
  - *Rather than responding according to the way it is currently referred to in the program.*
- This means that the compiler won't be always able to determine which method definition to use.
  - *The program won't know exactly what type of object is being sent the message until run-time.*
  - *This is termed **late binding**, binding a function call (message) to a particular function (method definition) as the program executes (at runtime).*

Polymorphism - C++ Pointers and Inheritance





```
Circle* c = new Circle(…);
Shape* s = c;    // what happens when we invoke the method ---
c->draw();   // will get the Circle class implementation
s->draw();  // will get the Shape class implementation, unexpected.
```

But a Circle IS A Shape, and we should be able to refer to a Circle as a Shape and still have the object respond properly according to the rules of Polymorphism.

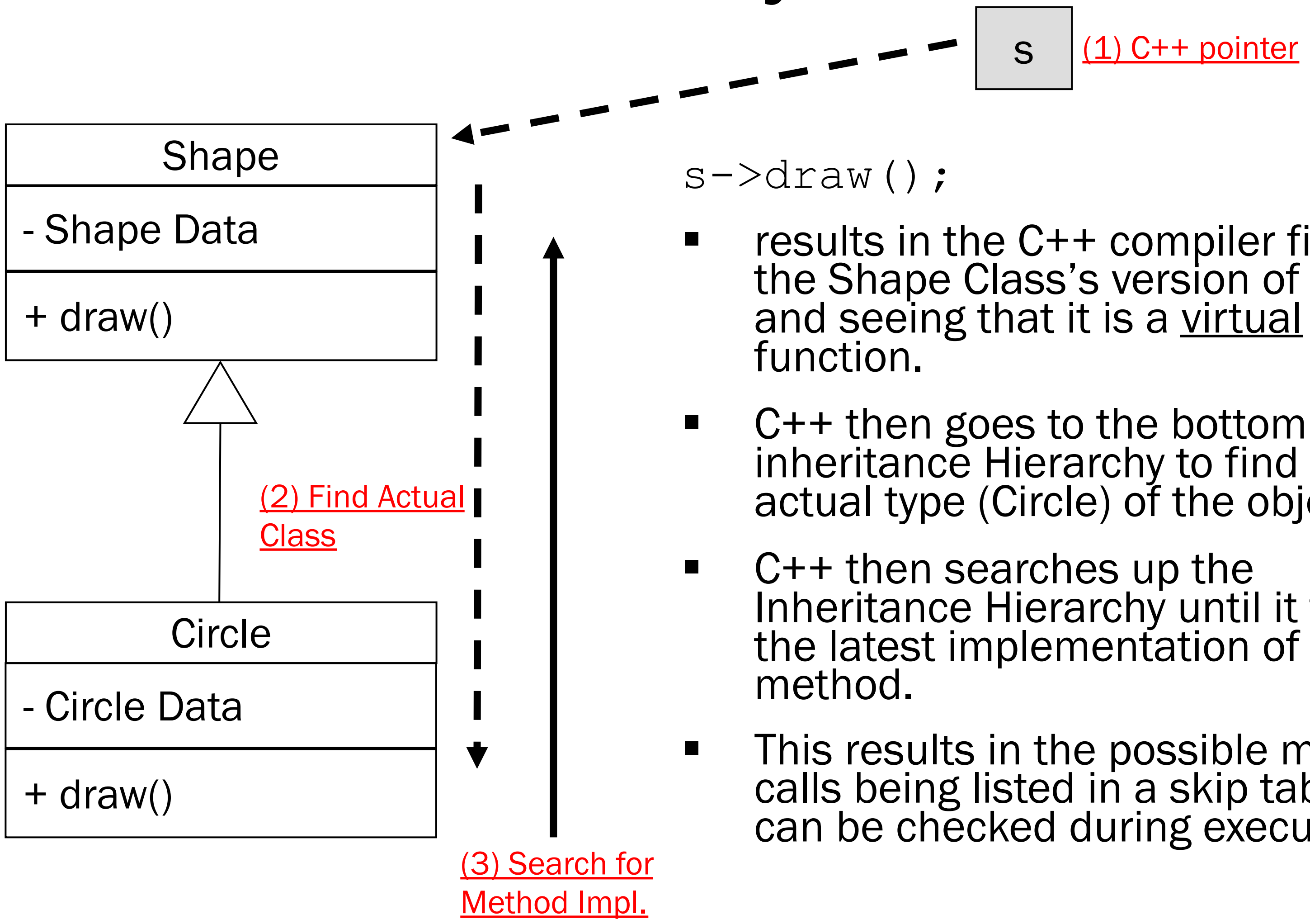
The C++ Answer - VIRTUAL

- C++ introduces the keyword *virtual* to allow differentiating between functions (methods) which are to be directly called and those which may be re-implemented in a derived class, even when the object is referred to as if it were a member of an ancestor class.

```
// in Shape class
virtual void draw() { … }
// in usage example
Circle* c = new Circle(…);
Shape* s = c;
s->draw();
```

Now we can access the Circle method draw().

How this fits into the Inheritance Hierarchy



```
s->draw();
```

- results in the C++ compiler finding the Shape Class's version of draw() and seeing that it is a <u>virtual</u> function.
- C++ then goes to the bottom of the inheritance Hierarchy to find the actual type (Circle) of the object.
- C++ then searches up the Inheritance Hierarchy until it finds the latest implementation of the method.
- This results in the possible method calls being listed in a skip table that can be checked during execution.

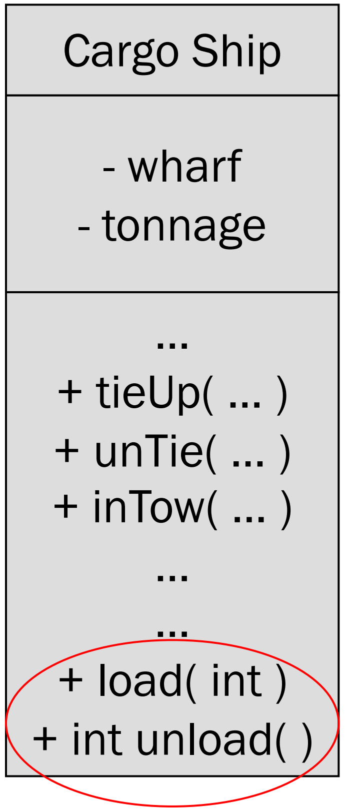
Specification vs Implementation of a Method

- There are two parts to every method and we now must be able to differentiate between them:
  - *<u>Specification</u>: Includes everything needed to be able to <u>call</u> a method as well as process whatever it might return*
    - The name of the method
    - The list of parameters that any call to the method must have
    - The type and contents of the answer that is returned
  - *<u>Implementation</u>: Everything needed to be able to actually execute the call*
    - The set of operations in their specific order, given any particular set of parameters, that are used in the call
- For example
  - *Circle is a <u>concrete</u> class because we know how to go about DOING all of the methods defined*
  - *That is, we have both specification and implementation of all of this class's methods.*

Polymorphism and Abstract Class

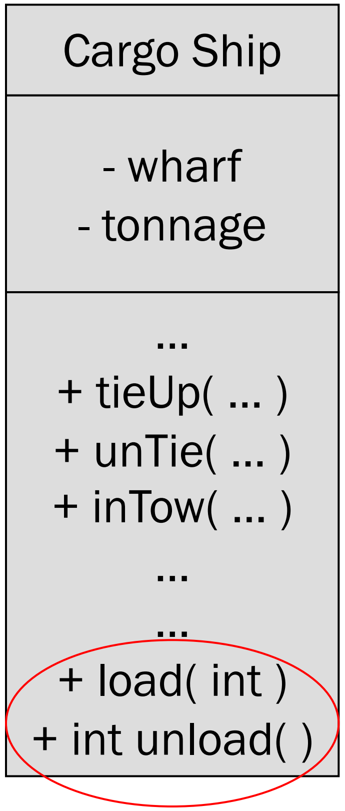
- There are times when we know how to specify a particular method but cannot determine (as yet) how to implement it
  - *We can know the name of the method*
  - *We can know the set of parameters we want to pass in*
  - *We know what type of answer we are expecting*
- But we don't know exactly HOW to go about doing the method in this general sense (e.g. 3 or 4 different types of Shapes).
- If we *can* SPECIFY any Method for a class but *cannot* IMPLEMENT the method (because things are too general at this time), then the class becomes an ABSTRACT Class, we cannot have objects belonging to this class but the specification of it is still VERY useful.

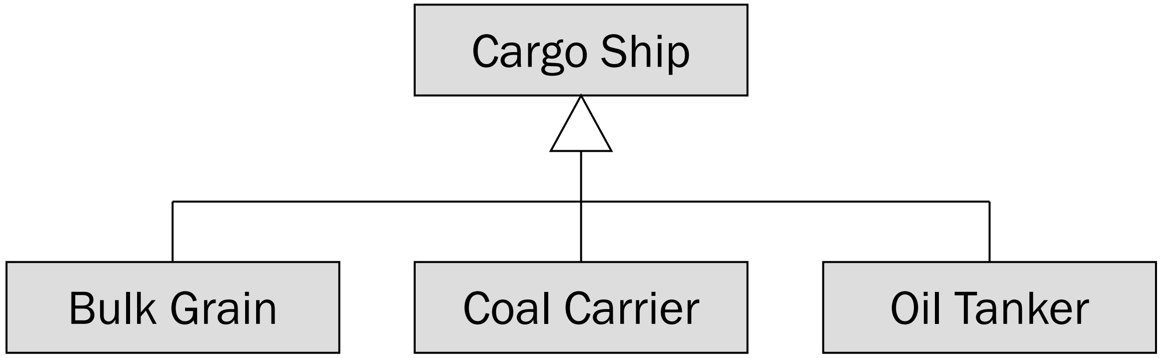
An Abstract Class – Cargo Ship



- No matter what sort of cargo the ship carries, it will always be tied up and untied in the same way, so these methods can be implemented in this class.
- No matter what sort of cargo, the ship will be taken in tow by a tug in the same way, and so this method can also be implemented.
- But the ways in which loading and unloading is done will depend on the cargo type and so these cannot be implemented and the class is *Abstract*

Concrete Classes - Cargo Ships





- Bulk Grain, Coal Carrier, Oil Tanker each contain their own implementations of
  - *`void load(int)`*
  - *`int unload()`*
- and are therefore concrete classes
- They can therefore be instantiated, e.g.

```
CargoShip* c = new OilTanker(…)
```

```
or
new BulkGrain(…) or
new CoalCarrier(…)
```

- but no matter which concrete class constructor is used a `CargoShip` pointer can still be used to reference it.

Abstract Classes in C++

- Abstract classes have at least one method that does not have an implementation.
- Therefore they cannot be instantiated
- An object of a derived concrete class (sub-class), once instantiated, can be referred to as a member of the abstract class (by a pointer or reference).
- Abstract classes can only be used as a step in the Inheritance Hierarchy for an object.

- But what about an array of CargoShips?
  - *In Java this is OK – the array is a separate object in its own right*
  - *In C++, this is not possible, you can only have pointers to CargoShips.*
  - *Remember that an array is just a simple container.*

Pure Virtual Functions in C++

- Shows that there is <u>no implementation</u> for the method in this class – so the class must be abstract.
  - *In class Cargo Ship*

```
virtual void load(int)= 0;
virtual int unload() = 0;
```

- Within class Shape

```
virtual void draw() = 0;
```

  - *Could be used if Shape were to be an abstract class with each sub-class now being required to have its own implementation of draw()*

Java Solution

- Every method is (in C++ terms) a “virtual” function and so does not need the keyword virtual.
- Java always uses late binding to resolve names of methods being called, Java objects are inherently polymorphic.
- No matter what type of reference is used in a Java program, when a method is called, Java will find the true type of the object, and work back up the inheritance hierarchy to find the appropriate definition of the method to be called.
- At the time that this decision was made, Java was purely interpreted and so the cost in performance was not that great – modern advances bring this under scrutiny (a little). Compiler code optimisation helps at times.

Compile-Time vs Run-Time Binding

Compile-time (early binding)

- Performed during code compilation.
- Method implementation is linked with the calls during the compilation.
- Faster program execution
  - *Code branch prediction methods, etc. can save a lot of execution time.*
- Implementation, may not depend on inheritance, e.g.,
  - *Method overloading*
  - *Template (C++)*
  - *Static/final/private methods (Java)*

Run-time (late binding)

- Performed during program execution.
- Method implementation is linked with the calls during the execution.
- Slower program execution
  - *Late bound method calls can vary according to the execution profile of the program.*
- Implementation, depends on inheritance hierarchy, e.g.,
  - *Method overriding*

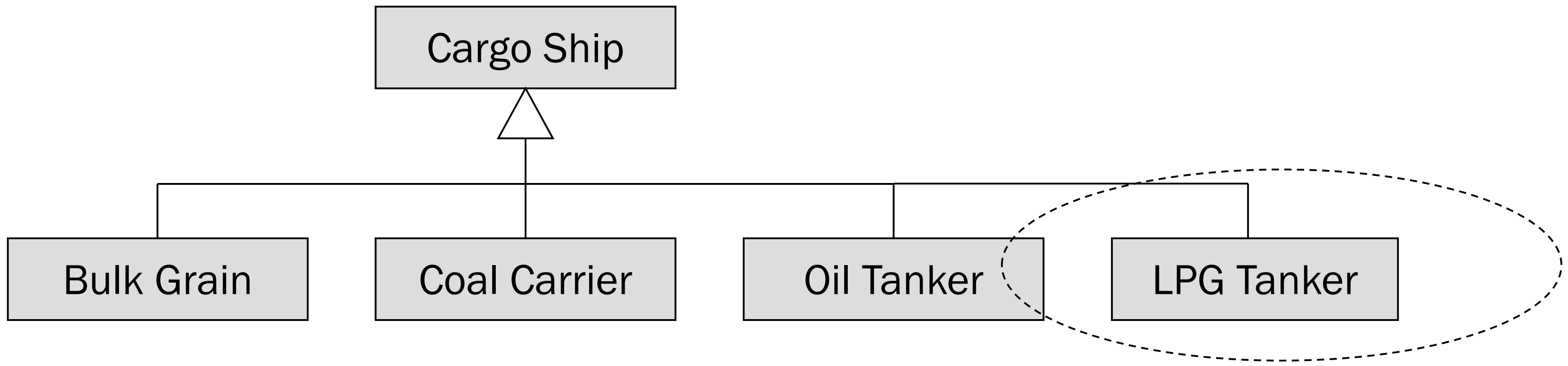
Extensibility is difficult

- In O-O modelling terms, it is very difficult to determine (forecast) how and where a piece of software is likely to be extended.
- If polymorphism and late-binding is so good, why wouldn't you always program like this?

ANSWER: Java does.

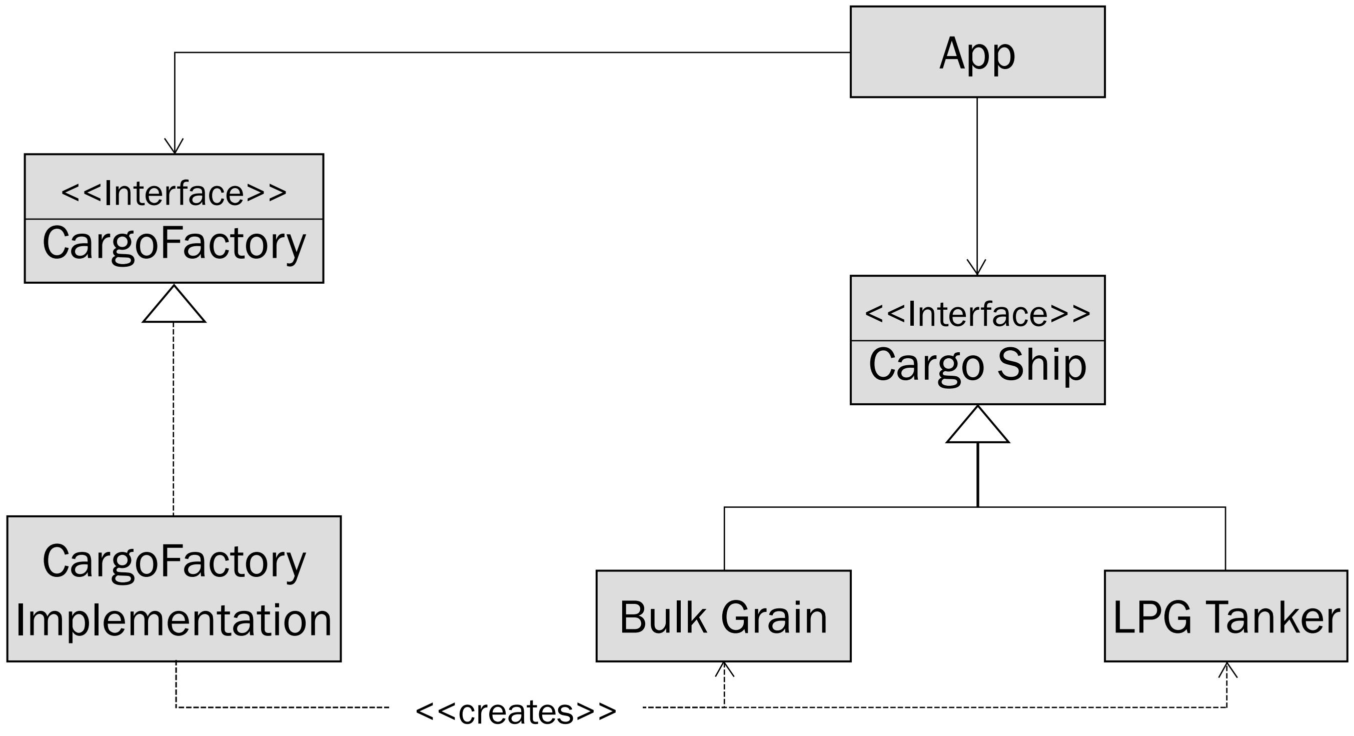
A language that always allows for late binding will always be ready to accept polymorphic-style extensions.

Adding a new class



- A new class with new method (& attribute) definitions
- Nothing about the other classes needs to be changed
- The only changes are where a particular type of cargo ship gets instantiated (the factory design pattern).
- How would the program be different without object-oriented programming?
- How would the program be different without polymorphism?

Factory Pattern



Cargo Ship Interface

```
public interface CargoShip {
    void load(int t);
}

public class BulkGrain implements CargoShip {
    public void load(int t) {
        System.out.println("Loading " + t + " tons on Bulk Grain.");
    }
}

public class LPGTanker implements CargoShip {
    public void load(int t) {
        System.out.println("Loading " + t + " tons on LPG Tanker.");
    }
}
```

Cargo Ship Factory Interface - 1

```
public interface CargoFactory {
    CargoShip makeBulkGrain();
    CargoShip makeLPGTanker();
}


public class CargoFactoryImplementation implements CargoFactory {
    public CargoShip makeBulkGrain() {
        return new BulkGrain();
    }

    public CargoShip makeLPGTanker() {
        return new LPGTanker();
    }
}
```

Cargo Ship Factory Interface - 2

```
public interface CargoFactory2 {
    CargoShip make(String cargoType) throws Exception;
}
```

```
public class CargoFactoryImplementation2 implements CargoFactory2 {
    @Override
    public CargoShip make(String cargoType) throws Exception {
        if (cargoType.equals("BULKGRAIN")) {
            return new BulkGrain();
        }
        else if (cargoType.equals("LPGTanker")) {
            return new LPGTanker();
        }
        else
            throw new Exception("Cargo ship factory cannot create " + cargoType);
    }
}
```

Polymorphism Advantages

- Allows each object to respond according to its own definition of what a particular method can do
  - *But what does this really mean?*
- When a new kind of object needs to be added into a system, it will mean that there will be a new class
  - *with a new definition of what the (polymorphic or virtual) method is supposed to do, as well as any extra attribute data needed.*
  - *the specification is already set out in the abstract class.*
  - *with its own constructor method*
  - *the other concrete classes do not have to be altered at all*
- A program can therefore be extended to include new capabilities (and new types of objects) without altering existing code (keeping alterations of code to a minimum).
- Polymorphism provides support for extensibility.

An Aside: C# Inheritance & Poly.

- Inheritance:
  - *When a derived class redefines a method in C#, the keyword overrides must be used.*
- Polymorphism:
  - *C# returns to the C++ strategy of making the base class method specification to carry the designation of virtual*
  - *The overrides designation for the defined/redefined methods under inheritance remains.*
  - *ie virtual .... overrides ...... overrides ....*

Pure Abstract Classes

- If a class contains ONLY pure virtual methods in C++ then it can be thought of as a pure abstract class.
    - *It contains NO implementation of any of its methods at all*
    - *It therefore contains ONLY specification of behaviour*
- The class exists as a source of behaviour specifications for inheritance.
- It may also be used as a basis for the reference of different types of objects according to a common reference specification (a C++ pointer or reference, or a Java reference).

Pure Abstract Classes and Attribute Data

- Attribute data in a pure abstract class can only be either protected (or public for class constants), as private data have no method implementations to make use of them, and once used as a basis of inheritance, any private data becomes inaccessible.
- It would be rare for an abstract class to have no method implementations but then to also have attribute data associated with it.
- Most pure abstract classes therefore have (virtual in C++) method specifications and no attribute data.
- This is basically what an ***interface*** is in Java – with Java adding some extra requirements on how they are used.

Java Interfaces

```
public interface Queue
{
   void enqueue(Object);
   Object dequeue();
   Object first();
   int size();
   boolean isEmpty();
}
```

Java does allow a class to be listed specifically as abstract, in which case not all methods need to be implemented.

- This only contains specification of behaviour
- Including it (effectively inheriting from it) into another class can therefore not run into the C++ diamond inheritance problem.
- Java stipulates that any class inheriting from an interface must provide implementations of ALL methods in the interface and consequently introduces the keyword ***implements*** to show this.

Standard Java Interfaces

- There are many groups of operations which might be useful on all types of objects, e.g.
  - *Comparable – where the objects can be ranked into a specific order.*
  - *Serializable – where the objects need to be "flattened" into a byte stream for writing to disk or sending over a network.*
  - *Cloneable – where an object needs to be copied rather than aliased.*
- There are standard ways of doing many of these operations that work for "most" objects and so implementations of them exist within the (abstract) Object class in Java, so that if they are not specifically implemented, then the standard (Object) one is used.

C++ “Interfaces”

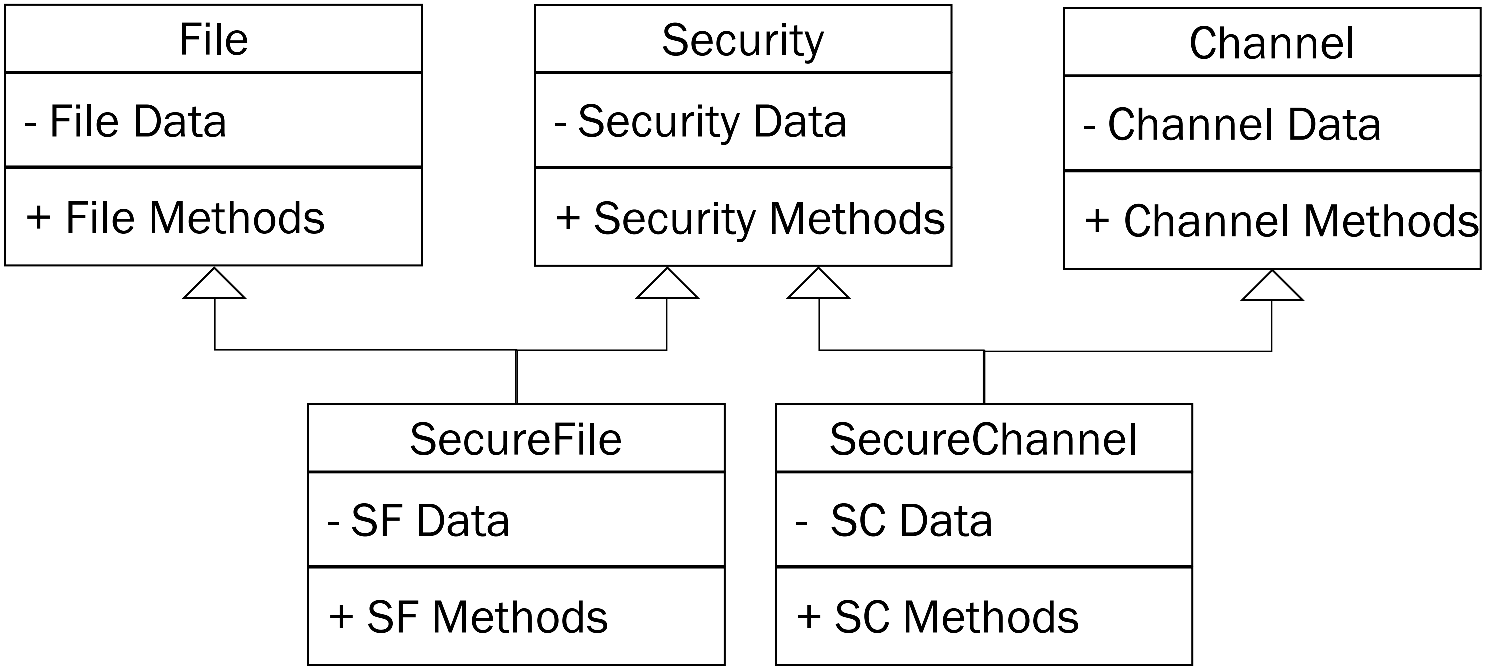
```
class Queue {
  public:
    virtual void enqueue(Object)=0;
    virtual Object dequeue()=0;
    virtual Object first()=0;
    virtual int size()=0;
    virtual boolean isEmpty()=0;
};
```

- You can do the same things as Java in O-O sense.
- In this case by use of Pure Abstract Classes
- In these cases, C++ does not have the diamond problem.
- C++ does not protect you from misuse of the facility.
  - *It does not restrict the way these classes are used and relies on good behaviour by the programmer.*

Aspects and Adjectival Types

- Interfaces are not a cure-all to the basic problems of C++ and diamond inheritance.
- It is possible to have an in-between case, where behaviour *and data* that might otherwise be thought of as an interface, have a possibly powerful use in O-O terms.
- These are the so-called “adjectival types” or “attribute types”.

Aspects and Adjectival Types



Aspects and Adjectival Types

- They have resulted in what is known as Aspect-Oriented Programming (referred to Cross-Cutting Concerns).
- C++ can effectively handle aspects via multiple inheritance, but without any specific protection from within the language to make sure that only aspects are included.
- Java requires a new language (e.g. AspectJ) to allow these capabilities to be implemented.

An Example: Rentable

Note that “Rentable” is an adjective.

```
Class Rentable {
private:
    -  String customerName;
    -  Date returnDate;
    -  Money costPerTime;
    .........
public:
    + void rentOut(Date);
    + Money returnItem(Date);
    .........
};
```

Rent A Car

- Add Rentable to a Car and get a Rent-A-Car.

```
Class RentACar : public Car, Rentable {
  …
  …
  …
};
```

- Rentable adds data and behaviour, but both the data and behaviour are guaranteed (by the programmer) not to interfere with the other base class (the noun), only to qualify it.
- The same facility can easily make a Rentable DVD for VideoEzy or a Rentable House for Dial Dowling R/E.
- The special thing to note is that there is no meaningful “IS A” relationship between a rental car, a rented DVD, or a rented house.

Cross-Cutting Concerns

- This is a major term used in Aspect-Oriented Programming
  - *We have seen with inheritance that data and methods appearing in a number of classes can be "Factored Out" to form a Base (Parent) class in an inheritance hierarchy*
- What about "Rentable" appearing in classes like Rented-Car, Rented-House and Rented-DVD?
  - *They appear in a collection of classes*
  - *BUT, there is no obvious relationship between the classes that can factor these to a higher point in the inheritance hierarchy*
  - *They "Cut-Across" the inheritance hierarchy*
  - *Houses, Cars and DVDs simply can't be made into a hierarchy*
- Aspects allow these to be factored "across" the inheritance hierarchy, implemented once, and then added back into relevant classes where needed.

Other Aspects

- The most popular Aspects to add into objects are
  - *Security* *for: SecureFile, SecureChannel*
  - *Transactions* *for: collecting a wide variety of actions into a single atomic action*
  - *Synchronisation*
  - *Monitoring* *for: measuring just about anything*
- But it is likely that wherever there is an adjective in your problem statement, that you may be able to make use of an Aspect-Style structure in your program, especially where the same adjective might be applied to a number of very different classes (classes where no inherent or meaningful inheritance relationship seem to exist).
- In some cases it will not be worth the extra effort, but as software systems scale – extra real benefits may appear.