Subclassing, Polymorphism, and Interfaces

V. Paúl Pauca

Department of Computer Science
Wake Forest University

CSC 331-631
Fall, 2013

Classes I

- A class defines a new type and a new scope
- Class definition

```
class Screen {
public:
    char get() const;

    typedef std::string::size_type index;

    Screen(): cursor(0), height(0), width(0) { }

    char get(index ht, index wd) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};

char Screen::get() const {
    return contents[cursor];
}
```

Classes II

- Class members?
- Constructor?
- Member functions?
- Function Overloading?
- Difference between declaring and defining a class?
- Defining a class object?

The Implicit `this` Pointer I

- Member functions have an extra implicit parameter, `this`
- `this` is a pointer to an object of the class type
- Bound to the object on which the member function is called
- When to use `this`?
- Concatenating a sequence of function calls

```
Screen myScreen;
...
myScreen.move(4,0).set('#');
```

The Implicit `this` Pointer II

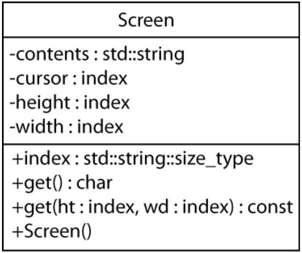
- Here is how

```
Screen& Screen::set(char c) {
    contents[cursor] = c;
    return *this;
}

Screen& Screen::move(index r, index c) {
    index row = r * width;
    cursor = row + c;
    return *this;
}
```

UML Class Diagram

- Type of Unified Modeling Language (UML) diagram
- Describes the structure of a class

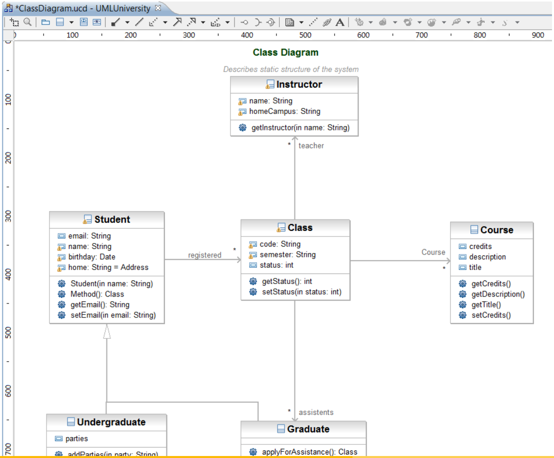
```
class Screen {
public:
    char get() const;

    typedef std::string::size_type index;

    Screen(): cursor(0), height(0), width(0) { }

    char get(index ht, index wd) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};

char Screen::get() const {
    return contents[cursor];
}
```



- -/+ indicate private/public

UML Diagram Tools

- Great deal of tools available, from simple to complex
- Simple (drawing): Visio, BOUML, ArgoUML, Eclipse, etc.
- Advanced (code generation): Visual Paradigm, Sparx, IBM Rational, etc



Subclassing

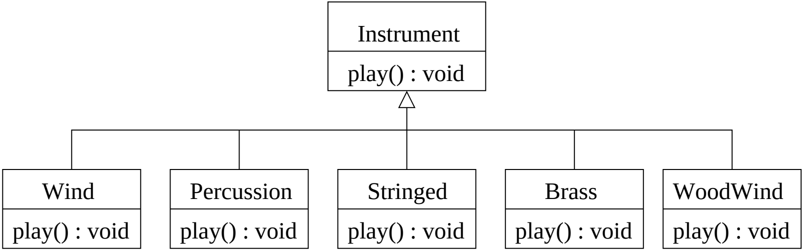
```
#include <iostream>
using namespace std;
enum note {middleC, Csharp, Eflat};

class Instrument {
public:
  void play(note) const {
    cout <<
     "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
```

```
#include <iostream>
#include "Instrument2.cpp"

using namespace std;

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute);
}
```

Virtual Functions

- In C++ define `play()` as `virtual` for polymorphic behavior
- In Java by default all base class instance methods are virtual
- Corresponding Unified Modeling Language (UML) diagram:

Subclasses


Instrument
play() : void
Wind
play() : void
Percussion
play() : void
Stringed
play() : void
Brass
play() : void
WoodWind
play() : void

Polymorphism I

- **Dynamic binding**: `x.foo()`, When a program decides at run time, which implementation of a given function, `foo()`, to invoke, based on the runtime type of object `x`
- `x` is an object $\Rightarrow$ then its type is static (decided at compile time)
- `x` is a reference $\Rightarrow$ then its type is dynamic (decided at run time)
- Virtuals are resolved at run time only if the call is made through a reference (or pointer)
- Nonvirtual calls are resolved at compile time

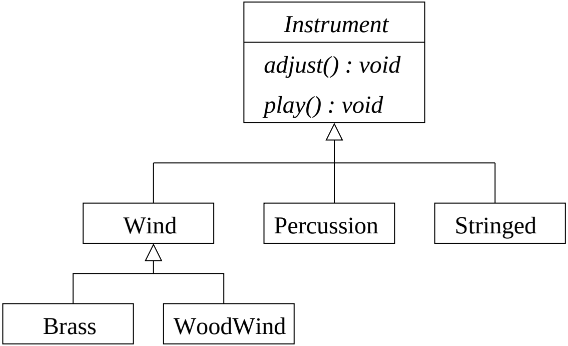
Polymorphism II

Here `i.play(middleC)` is polymorphic

```
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

Abstract Classes and Interfaces

`Instrument` as an abstract class



Advantages

- `Instrument` defines a common API for all subclasses
- It also separates the interface from the implementation

Abstract Classes and Interfaces

- In C++, `adjust()` and `play()` are defined as *pure* virtual functions.

```
class Instrument {
public:
   // Pure virtual functions:
   virtual void play(note) const = 0;
   virtual void adjust(int) = 0;
};
```

- In Java, an abstract class or method is defined with the `abstract` modifier.

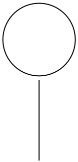
Abstract Classes and Interfaces

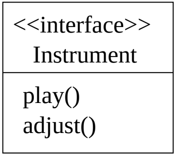
- Better to use interfaces in Java rather than abstract classes.

```java
public interface Instrument {
  public void play(Note note);
  public void adjust(int i);
};
```

- Pure abstract class or interface in UML:

Instrument





Abstract Classes and Interfaces

- Providing a particular interface / implementation

C++

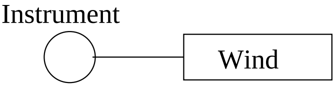
```cpp
class Wind :
  public Instrument {
public:
  void play(note) const {
    cout <<
     "Wind::play" << endl; }
  void adjust(int) {}
};
class Percussion :
  public Instrument {
public:
  void play(note) const {
    cout <<
     "Percussion::play"
     << endl; }
  void adjust(int) {}
};
```

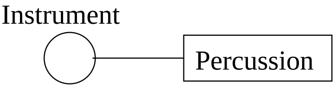
Java

```java
public class Wind
   implements Instrument {
  public void play(Note note) {
    System.out.println(
      "Wind::play"); }
  public void adjust(int i) {}
}

public class Percussion
   implements Instrument {
  public void play(Note note) {
    System.out.println(
      "Percussion::play"); }
  void adjust(int i) {}
}
```

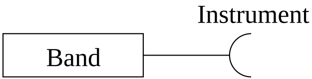
Abstract Classes and Interfaces

- A provided interface in UML





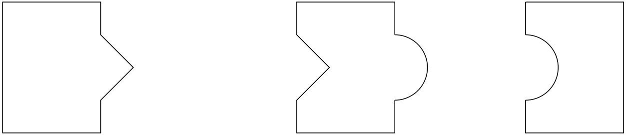
- A required interface



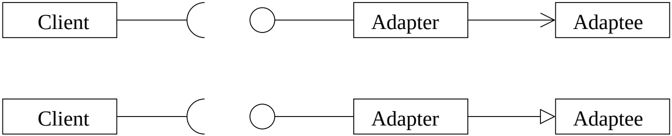

```
class Band {
   Instrument* inst;
public:
   Band() {}
   void addInstrument(Instrument) {}
};
```

Adapter Pattern

- A wrapper that provides a new interface for an existing component.



- The Object and Class Adapter Patterns



Example 1: From Head First Design Patterns

- Duck and Turkey interfaces and provider classes

```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck
  implements Duck {
    public void quack() {
        System.out.println(
          "Quack");
    }
    public void fly() {
        System.out.println(
          "I'm flying");
    }
}
```

```
public interface Turkey {
    public void gobble();
    public void fly();
}

public class WildTurkey
  implements Turkey {
    public void gobble() {
        System.out.println(
          "Gobble gobble");
    }
    public void fly() {
        System.out.println(
          "I'm barely flying");
    }
}
```

Example 1: From Head First Design Patterns

- The Turkey adapter

```
public class TurkeyAdapter implements Duck {
   Turkey turkey;
   public TurkeyAdapter(Turkey t) {
      turkey = t;
   }
   public void quack() {
      turkey.gobble();
   }
   public void fly() {
      for(int i=0; i<5; i++)
         turkey.fly();
   }
}
```

Example 1: From Head First Design Patterns I

- The Client

```
public class DuckTestDrive {
   public static void main(String[] args) {
      MallardDuck duck = new MallardDuck();
      WildTurkey turkey = new WildTurkey();
      Duck TurkeyAdapter = new TurkeyAdapter(turkey);

      System.out.println("Turkey says...");
      turkey.gobble(); turkey.fly();

      System.out.println("Duck says...");
      duck.quack(); duck.fly();

      System.out.println("TurkeyAdapter says...");
      testDuck(turkeyAdapter);
   }

   static void testDuck(Duck d) {
     d.quack(); d.fly();
   }
}
```

Example 1: From Head First Design Patterns II

- The Output

```
>> java DuckTestDrive

Turkey says...
Gobble gobble
I'm barely flying

Duck says...
Quack
I'm flying

TurkeyAdapter says...
Gobble gobble
I'm barely flying
I'm barely flying
I'm barely flying
I'm barely flying
I'm barely flying
```

Example 1: From Head First Design Patterns

- Which is the client and which is the adaptee?

  Draw the Turkey Adapter example in UML.

Example 2: From Head First Design Patterns

- Legacy code build around the `Enumeration` interface

<<inteface>> Enumeration
hasMoreElements() nextElement()

- But newer code should use only `Iterator`s

<<inteface>> Iterator
hasNext() next() remove()

- Rather than changing all the legacy code, use an adapter pattern to make an `Enumeration` behave like an `Iterator`.

Example 2: From Head First Design Patterns

- The adapter class makes all providers of the `Enumeration` interface accessible to the newer code.