Programming Languages (CS 421/521): M. Allen, 22 Apr. 19

Program Composition and Decomposition

Programming languages have long allowed us to handle complexity using procedural decomposition:
1. Start with a function to compute on data
2. Break it up into simpler functions
3. Stop at point that each function is written in basic constructs of our language

```java
sort(int[] arr)
compare(int x, int y)
swap(int[] arr, int x, int y)
```

Data Abstraction

Take the idea further: rather than abstracting away how some particular function/method works, we abstract away the details of how some data type works

- Encapsulation: grouping a collection of related constants and variables, or a variety of types, into a single object
- As with functional abstraction, we will be concerned with the interface to a type of object:
  1. How to create them
  2. What they can do
  3. What can be done to them

We can also simplify things by abstraction
Given a set of functions, we are mainly concerned with their interfaces: what they do, and what is necessary to run them properly, not necessarily details of how they do it
We abstract from the details of code for a function or method, and focus mainly on their inputs and outputs
Objects and Classes

- We want to decompose a program into a set of objects that communicate with one another.
- An object has its own particular functions that it can perform.
- Within an object, we may use functional decomposition to achieve sub-tasks.
- A class defines a general, abstract data type, containing all its own data-type elements, and all the operations (functions/methods) that define the class of object.

History of a Concept

- In the 1960's, Norwegian researchers O.-J. Dahl and K. Nygaard developed Simula, the first object-oriented language.
- Based on Algol, but with many new ideas, introducing concepts of a class, subclass, inheritance, abstract/virtual functions and overriding, garbage collection, etc.

```plaintext
begin
  class Animal;
  virtual: procedure hello is procedure hello;
  begin
    Animal class Dog(name);
    text name;
    begin
      procedure hello;
      begin
        outtext(name);
        outtext(" says woof!");
      end;
    end;
  end;
end;
```

Object Oriented Languages

- An object oriented language will generally support:
  1. Data encapsulation and hiding: limit scope and visibility of internal state variables and utility functions.
  2. Code inheritance: re-use of functionality by derived classes.
  3. Functional abstraction and dynamic dispatch (polymorphism): objects that override methods will determine the behavior of that method for themselves at runtime.

- Modern examples include:
  - C++
  - Java
  - Scala
  - OCaml
  - Ada
  - Python
  - Eiffel
An Example: Java

1. Encapsulation and hiding are supported by:
   - Access control (public, private, protected)
   - Static scope rules (local, global variables)
2. Inheritance is supported by:
   - Class extension (extends)
   - Interface implementation (implements)
3. Abstraction and dynamic dispatch are supported by:
   - Overriding of functions (and operators) in subclasses
   - Methods/classes that are abstract
   - Type conformance and polymorphism

Questions of Language Design

- The basic concepts supported by Java are supported by many other languages
- However, many choices can be made within an OO framework:
  - What can a derived subclass access from a base class?
  - Can a class have multiple direct superclasses?
  - Can a class be parameterized?
  - How does polymorphism actually work?
  - Is it done at compile time or run-time?

Inheritance and Access

- In Java, all aspects of a superclass are inherited by an extending subclass, though only public and protected elements can be directly accessed
- Overriding methods replace the superclass version
- Within the class, parent methods can be accessed (super), but not from outside the class, hiding the parent version from users of the child
- Access levels (public, etc.) can’t be changed by the child class
- C++ makes some of these decisions differently
  - A subclass can hide anything from the parent it wants
  - It can inherit from the parent in a public way, keeping everything at the same access level, or in a private way, restricting all parent methods to internal use only
  - It can also selectively reveal some of the parent methods

Overriding and Dynamic Dispatch

- When a method in a subclass overrides (redefines) a corresponding super-class method, questions can arise as to which version of the method should be used at different points in the code
- Problems can arise for common type conformance systems, since objects of a derived class may be referred to either as base class type or as derived class type
Overriding and Dynamic Dispatch

- Java uses **dynamic method binding** by default
  - The type of a **variable** does not determine the behavior of a method
  - The actual **object** bound to that variable at run-time decides this
  - Different versions of the method may be called for the same variable at various different points in time
  - Compiler only guarantees that some version of the method will be present
  - Programmers can overcome this: **final** methods can’t be over-ridden

- C++ uses **static method binding** by default
  - The type of a variable **does** determine the version of a method used
  - At compile time, bindings of method calls to actual code are generated (resulting in some run-time efficiencies)
  - Programmers can overcome this: **virtual** methods are abstract and must be over-ridden, and run-time checks are used to bind polymorphically

Parameterized (Generic) Types in Java

- In Java 1.4 and earlier:
  - All collections store and return **Object** type
  - Usually must **cast** for the specific type being used

```java
Vector vec = new Vector();
for (int i = 0; i < 10; i++)
    vec.add(new Integer(i));

int sum = 0;
for(int i = 0; i < vec.size(); i++)
    sum += ((Integer) vec.get(i)).intValue();
```

- Vector<Integer> v1 = new Vector<Integer>(2);
  - System.out.println(v1.get(0));
  - System.out.println(v2.get(0));

Parameterized (Generic) Types in Java

- Many **OO Languages** support generic types which can be specialized by providing the "missing" information when necessary
- Classes and their methods can take parameters that define what their various instances will actually take in and return
- Java 1.5 and later, C++, Ada, and Scala all provide this mechanism

```java
public class Vector<T>
{
    private ArrayList<T> data;
    private int size;

    public Vector(int cap) {
        data = new ArrayList<T>(cap);
        size = 0;
    }

    public T get(int i) {
        return data.get(i);
    }

    public void add(T d) {
        data.add(d);
    }
}
```

```java
Vector<Integer> v1 = new Vector<Integer>(2);
Vector<String> v2 = new Vector<String>(2);

v1.add("Hello");
System.out.println(v1.get(0));
v2.add("Hello");
System.out.println(v2.get(0));
```

Generics and Subtyping

- Generics introduce complex sub-typing issues

```java
LinkedList<String> ls = new LinkedList<String>();
ls.insert("Hello", 0);
ls.start();
String s = ls.getCurrent();
System.out.println(s);
```

```
LinkedList<Object> lo = ls;
LinkedList<Object> lo = (LinkedList<Object>) ls;
```

```
String s = lo.getCurrent();
s = (String) lo.getCurrent();
System.out.println(s);
```

These lines fail! (WHY?)

Also fails! (WHY?)
Single vs. Multiple Inheritance

- In Java, every class has **only one** direct super-class
- The inheritance hierarchy is always a tree
- We can inherit along a chain of single classes

- In C++, inheritance from **multiple** super-classes allowed
- There is no "root class" like Object
- All classes without a super-class root their own hierarchy
- More powerful than Java but also can lead to difficulties

### C++ Multiple Inheritance Example

```cpp
#include <string>
using namespace std;

class NamedObject {
public:
    NamedObject(const string& str) { name = str; }
    string getName() { return name; }
private:
    string name;
};

class Locatable2DObject {
public:
    Locatable2DObject(double xin, double yin) {
        x = xin;
        y = yin;
    }
    double getX() { return x; }
    double getY() { return y; }
    void setX(double xin) { x = xin; }
    void setY(double yin) { y = yin; }
private:
    double x, y;
};

class NamedShape : Locatable2DObject, NamedObject {
public:
    NamedShape(double x, double y, string n) :
        Locatable2DObject(x,y),
        NamedObject(n){}
};
```

### C++ Example: Method Collision

```cpp
class A {
public:
    A(long id): ident(id) {} // public
    int getID() { return ident; } // private
private:
    long ident;
};

class B {
public:
    B() {} // public
    int getID() { return 500; } // private
};

class NewClass : A, B {
public:
    NewClass(long id): A(id), B() {} // correct
};
```

### This Week

- **Topic**: Object-Oriented Languages (Text, chapter 10)
- **Meet**: usual schedule
- **Homework 04**: due Friday, 26 April, 5:00 PM
- **Office Hours**: Wing 210
  - Monday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 8:30 AM – 10:00 AM