Fundamental Concepts in OOP

- **Encapsulation**
  - Data Abstraction
  - Information hiding
  - The notion of class and object

- **Inheritance**
  - Code reusability
  - Is-a vs. has-a relationships

- **Polymorphism**
  - Dynamic method binding
Fundamental Concepts in OOP

- Encapsulation
  - Data Abstraction
  - Information hiding
  - The notion of class and object

- Inheritance
  - Code reusability
  - Is-a vs. has-a relationships

- Polymorphism
  - Dynamic method binding

Inheritance

- Encapsulation improves code reusability
  - Abstract Data Types
  - Modules
  - Classes

- However, it is generally the case that the code a programmer wants to reuse is close but not exactly what the programmer needs

- Inheritance provides a mechanism to extend or refine units of encapsulation
  - By adding or overriding methods
  - By adding attributes
Inheritance

Notation

Java.awt.Dialog

Base Class
(or Parent Class
or Superclass)

Is-a relationship

Java.awt.FileDialog

Derived Class
(or Child Class
or Subclass)

Inheritance

Subtype

Java.awt.Dialog

Base Class

Is-a relationship

Java.awt.FileDialog

Derived Class

• The derived class has all the members (i.e. attributes and methods) of the base class
  – Any object of the derived class can be used in any context that expect an object of the base class
  – fp = new FileDialog() is both an object of class Dialog and an object of class File Dialog
Method Binding

```cpp
class person {
  ...
}
class student : public person {
  ...
}
class professor : public person {
  ...
}

student s;
professor p;

person *x = &s;
person *y = &p;

void person::print_mailing_label () { ... }
...

x->print_mailing_label ();    // student::print_mailing_label (s)
p->print_mailing_label ();    // professor::print_mailing_label (p)
```

Method `print_mailing_list` is polymorphic

Results depend on the binding: static or dynamic

Print_mailing_label redefined for student and professor classes

---

Method Binding

Static and Dynamic

- **In static method binding**, method selection depends on the type of the variable `x` and `y`
  - Method `print_mailing_label()` of class `person` is executed in both cases
  - Resolved at compile time

- **In dynamic method binding**, method selection depends on the class of the objects `s` and `p`
  - Method `print_mailing_label()` of class `student` is executed in the first case, while the corresponding methods for class `professor` is executed in the second case
  - Resolved at run time
Polymorphism and Dynamic Binding

- The is-a relationship supports the development of generic operations that can be applied to objects of a class and all its subclasses
  - This feature is known as polymorphism
  - E.g. paint() method is polymorphic (accepts multiple types)

- The binding of messages to method definitions is instance-dependent, and it is known as dynamic binding
  - It has to be resolved at run-time
  - Dynamic binding requires the virtual keyword in C++
  - Static binding requires the final keyword in Java

Dynamic Binding Implementation

- A common implementation is based on a virtual method table (vtable)
  - Each object keeps a pointer to the vtable that corresponds to its class

```cpp
class foo {
    int a;
    double b;
    char c;
    public:
        virtual void k ( ... 
        virtual int l ( ... 
        virtual void m ();
        virtual double n( ... 
    };
```

```
E  
F  
foo's vtable
  k
  l
  m
  n
  Code for c
```

Felix Hernandez-Campos
Dynamic Binding Implementation

- Given an object of class foo, and pointer f to this object, the code that is used to invoke the appropriate method would be

to call f->m():

\[
\begin{align*}
  & \text{r1 := f} \quad \text{(this (self))} \\
  & \text{r2 := *r1} \quad \quad \quad \quad \text{-- vtable address} \\
  & \text{r2 := *(r2 + (3-1) \times 4)} \quad \quad \text{-- assuming 4 = sizeof(address)} \\
  & \text{call *r2} \quad \quad \quad \quad \text{(polymorphic) method invocation}
\end{align*}
\]

Dynamic Binding Implementation

Simple Inheritance

- Derived classes extend the vtable of their base class
  - Entries of overridden methods contain the address of the new methods

```cpp
class bar : public foo {
  int w;
  public:
    void m(); // override
    virtual double s() { ... }
    virtual char* t() { ... }
} B;
```

```
B                   bar's vtable
k                  Code for bar's m
l                  Code for foo's m
n                  Code for foo's m
a                  Code for bar's s
b                  Code for bar's s
c                  Code for bar's t
v                  Code for bar's t
```
Dynamic Binding Implementation
Multiple Inheritance

- A class may derive from more than one base class
  - This is known as multiple inheritance
- Multiple inheritance is also implemented using vtables
  - Two cases
    - Non-repeated multiple inheritance
    - Repeated multiple inheritance

Dynamic Method Binding
Non-Repeated Multiple Inheritance
**Dynamic Method Binding**

**Non-Repeated Multiple Inheritance**

- The view of this must be corrected, so it points to the correct part of the objects
  - An offset $d$ is use to locate the appropriate vtable pointer
    - $d$ is known at compile time

```plaintext
to call my_student.debug.print:

\[\begin{align*}
  r1 & := \text{my_student} & \quad & \text{-- student view of object} \\
  r1 & := r1 + d & \quad & \text{-- gp_list_node view of object} \\
  r2 & := *r1 & \quad & \text{-- address of appropriate vtable} \\
  r3 & := *(r2 + (3-1) \times 8) & \quad & \text{-- method address} \\
  r2 & := *(r2 + (3-1) \times 8 + 4) & \quad & \text{-- this correction} \\
  r1 & := r1 + r2 & \quad & \text{-- this} \\
  \text{call } *r3
\end{align*}\]
```

---

**Dynamic Method Binding**

**Repeated Multiple Inheritance**

- Multiple inheritance introduces a semantic problem: method name collisions
  - Ambiguous method names
  - Some languages support inherited method renaming (e.g. Eiffel)
  - Other languages, like C++, require a reimplementation that solves the ambiguity
  - Java solves the problem by not supporting multiple inheritance
    - A class may inherit multiple interfaces, but, in the absence of implementations, the collision is irrelevant
Reading Assignment

• Scott
  – Read Sect. 10.4
  – Read Sect. 10.5 intro and 10.5.1