Class Hierarchy Analysis

What's the Problem?

```
List<String> xs = …;
int len = xs.size();
```

- How is this method invocation **compiled**?
- Can we determine set of **target methods**?
- What **advantages** if we can?

A Tale of Two Invocations

- **Direct** Invocation
  - E.g. calling **static** method in Java
  - E.g. calling a **non-virtual** method in C++
  - **Translates** to e.g. an x86 "**`call <addr>`**" instruction
  - Is considered relatively **efficient**

- **Indirect** Invocation
  - E.g. calling an **instance** method in Java
  - E.g. calling a **virtual** method in C++
  - E.g. translates to e.g. an x86 "**`call <register>`**" instruction
  - Is considered relatively **inefficient**

Virtual Method Table (a.k.a. “vtable”)

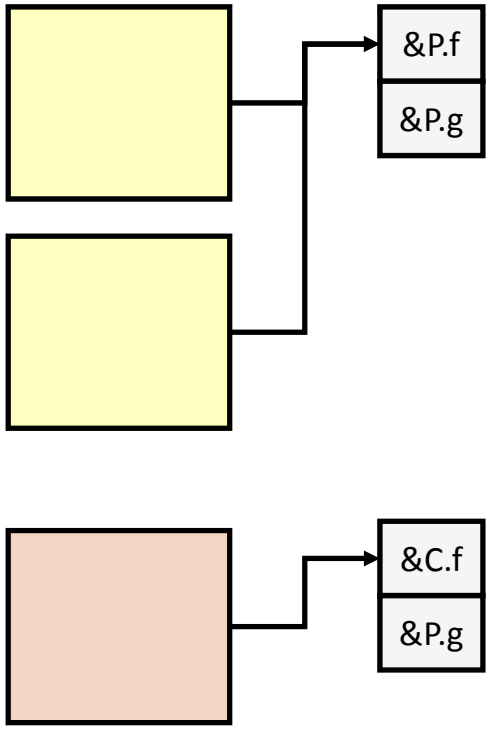
“Whenever a class defines a virtual function (or method), most compilers add a hidden member variable to the class that points to an array of pointers to (virtual) functions called the **virtual method table**. These pointers are used at runtime to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class. “

-- Wikipedia

Virtual Method Table (a.k.a. “vtable”)

```
class P {
  void f(int x) { … }
  void g(int x) { … }
}

class C extends P {
  void f(int x) { … }
}
```



Class Hierarchy Analysis

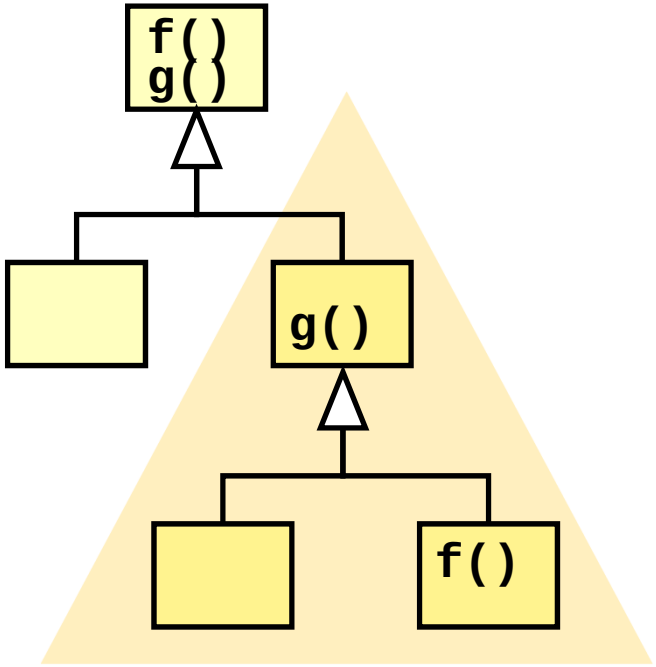
```
class P {
  void f(int x) { … }
  void g(int x) { … }
}

class C extends P {
  void f(int x) { … }
}
```

- Given a variable x of type P …
  - What set of methods could be called from "**`x.f()`**" or "**`x.g()`**"?
  - Can assume **Closed World Assumption** (why?)

Class Hierarchy Analysis

- The “cone” is a **core concept**



  - Set of classes **below and including** a given class
  - With this, can determine set of **overriding** methods

Flow Analysis

```
int getArea(int x, int y, int w) {
  return new Square(x,y,w).area()
}
```

```
String toString(boolean b, Collection xs) {
  List ls;
  if(b) { ls = new ArrayList<>(xs); }
  else { ls = new LinkedList<>(xs); }
  return ls.toString();
}
```

- What do we **know** about these invocations?