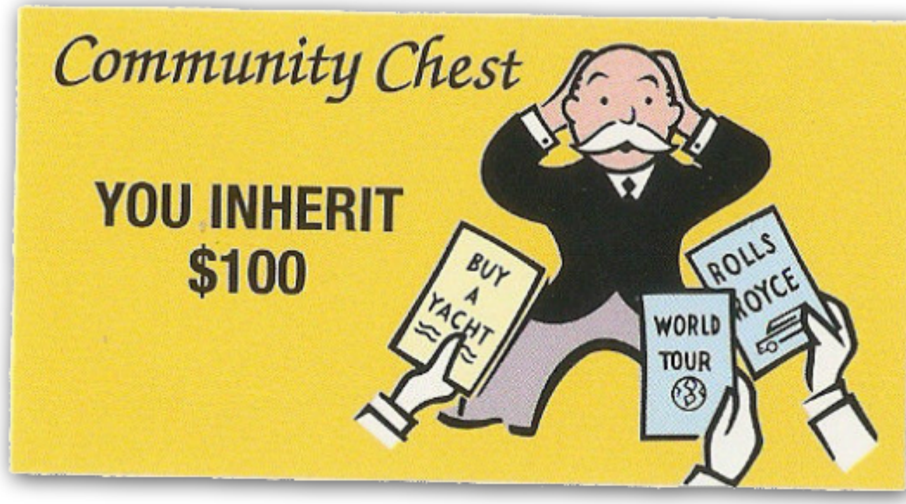
CS 106X

Lecture 26: Inheritance and Polymorphism in C++

Monday, March 13, 2017



Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

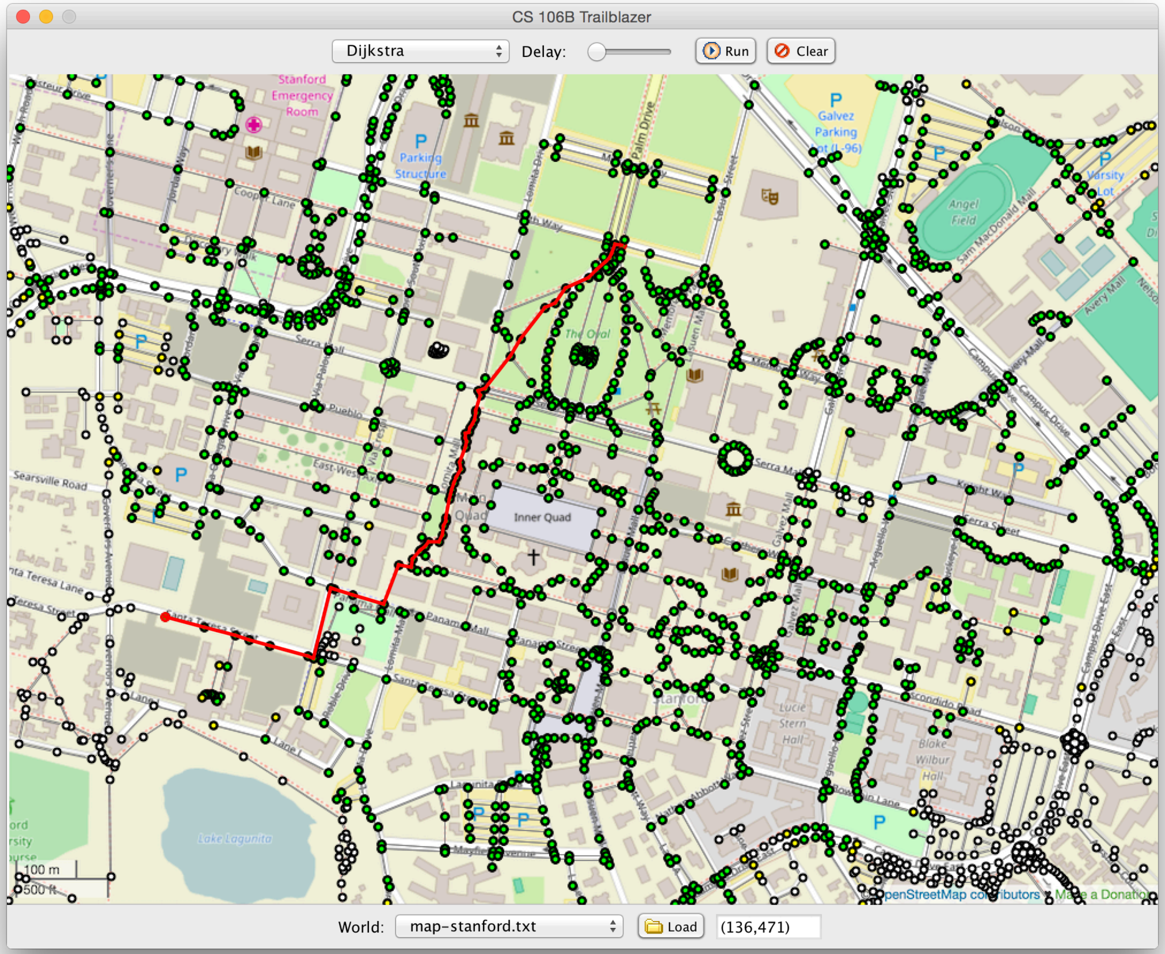
reading:
Programming Abstractions in C++, Chapter 19

Today's Topics

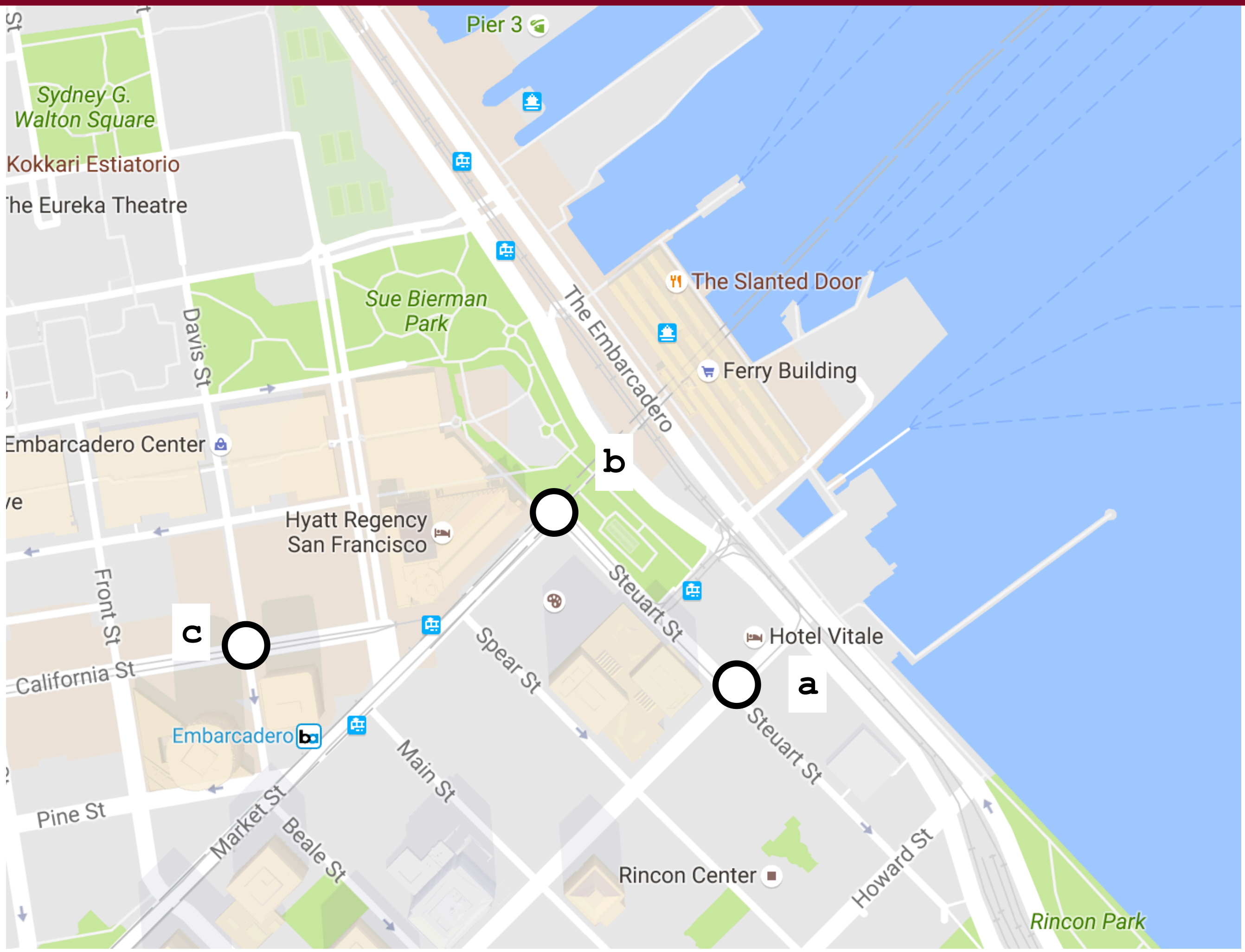
- Logistics
  - Final Exam prep online: http://web.stanford.edu/class/cs106x/handouts/final.html
  - Final exam is on Monday, March 20th at 8:30am.
  - Course evaluations now open on Axess

- A bit more on A*
- Inheritance and Polymorphism in C++

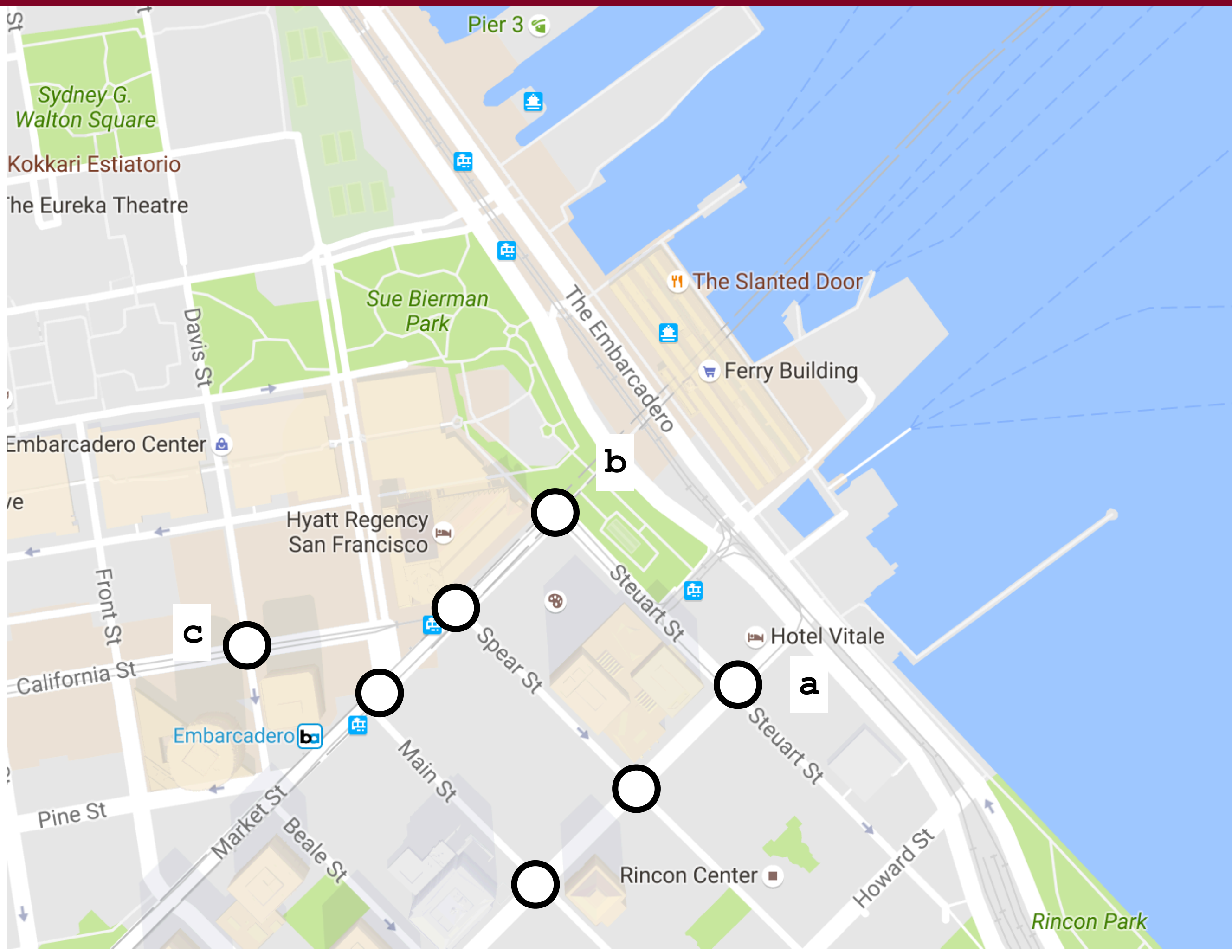
Trailblazer



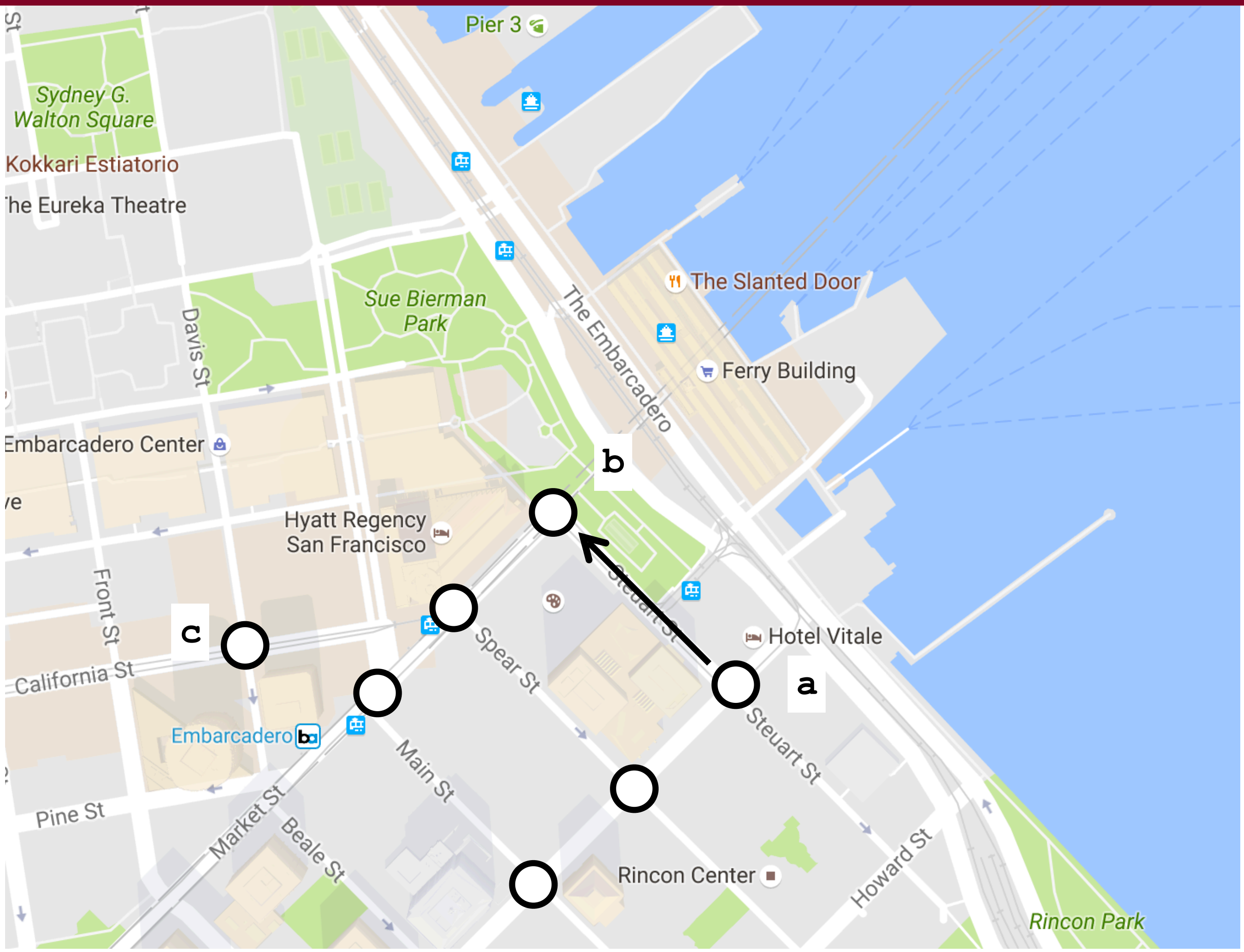
Road Map Node



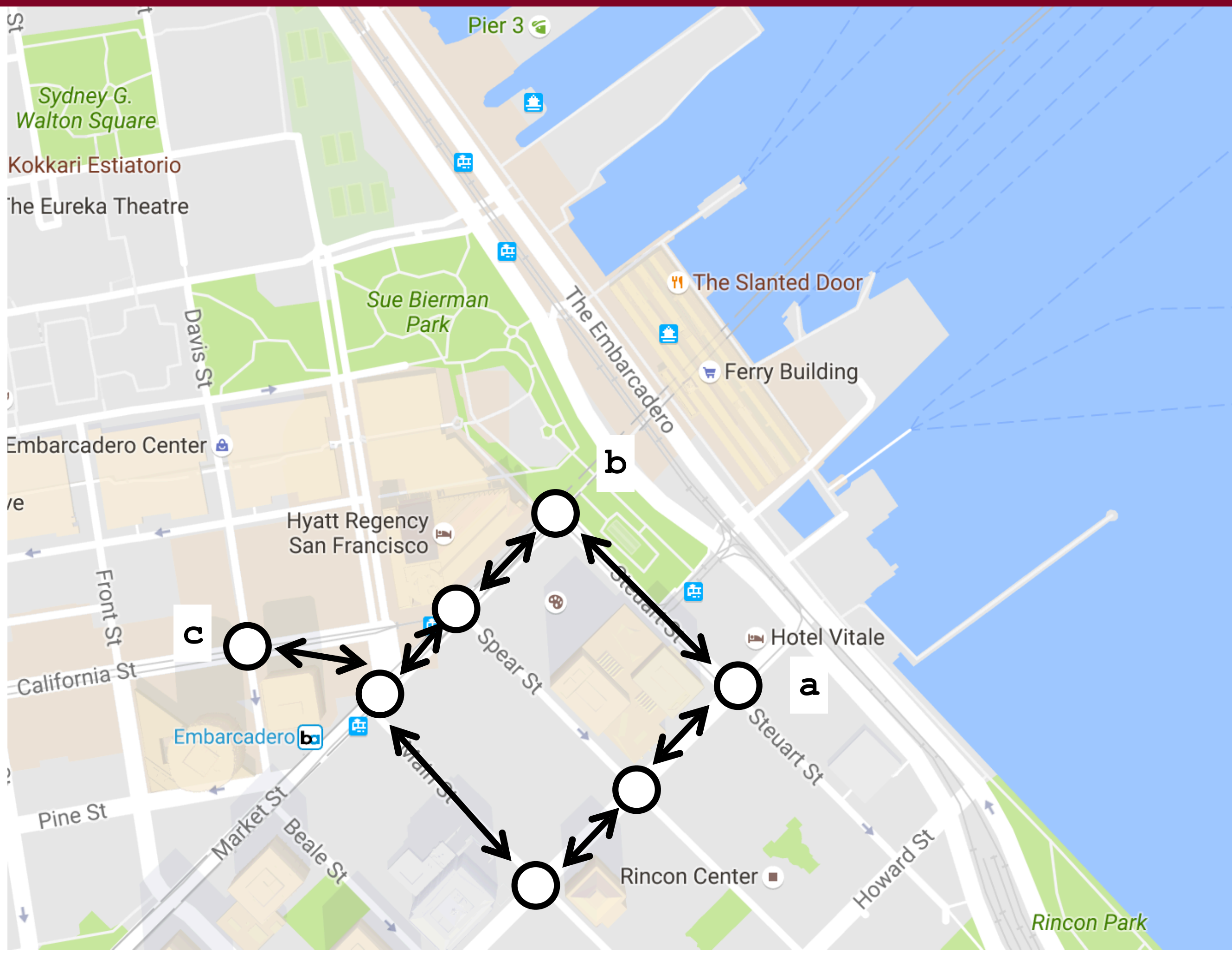
Road Map Node



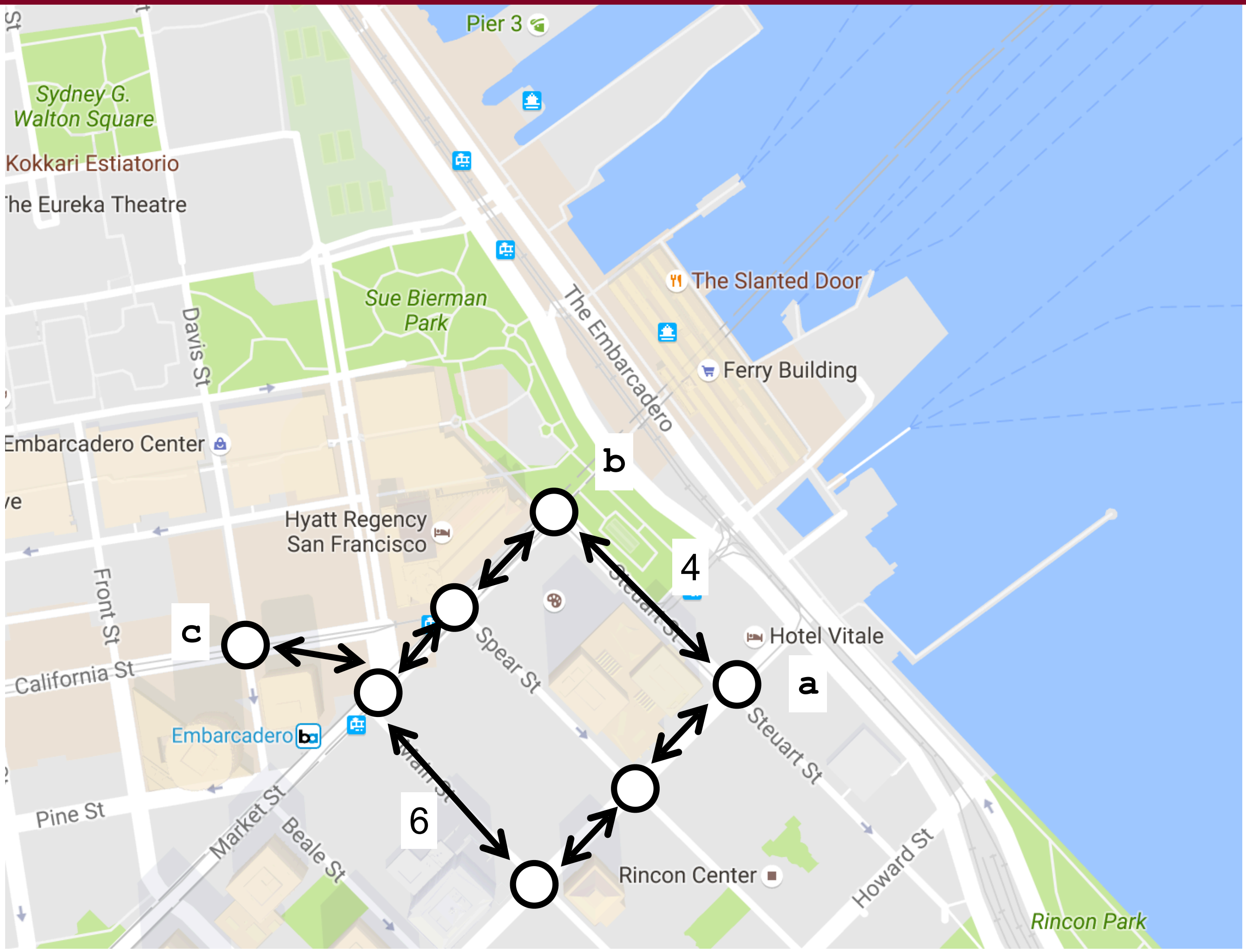
Road Map Edge



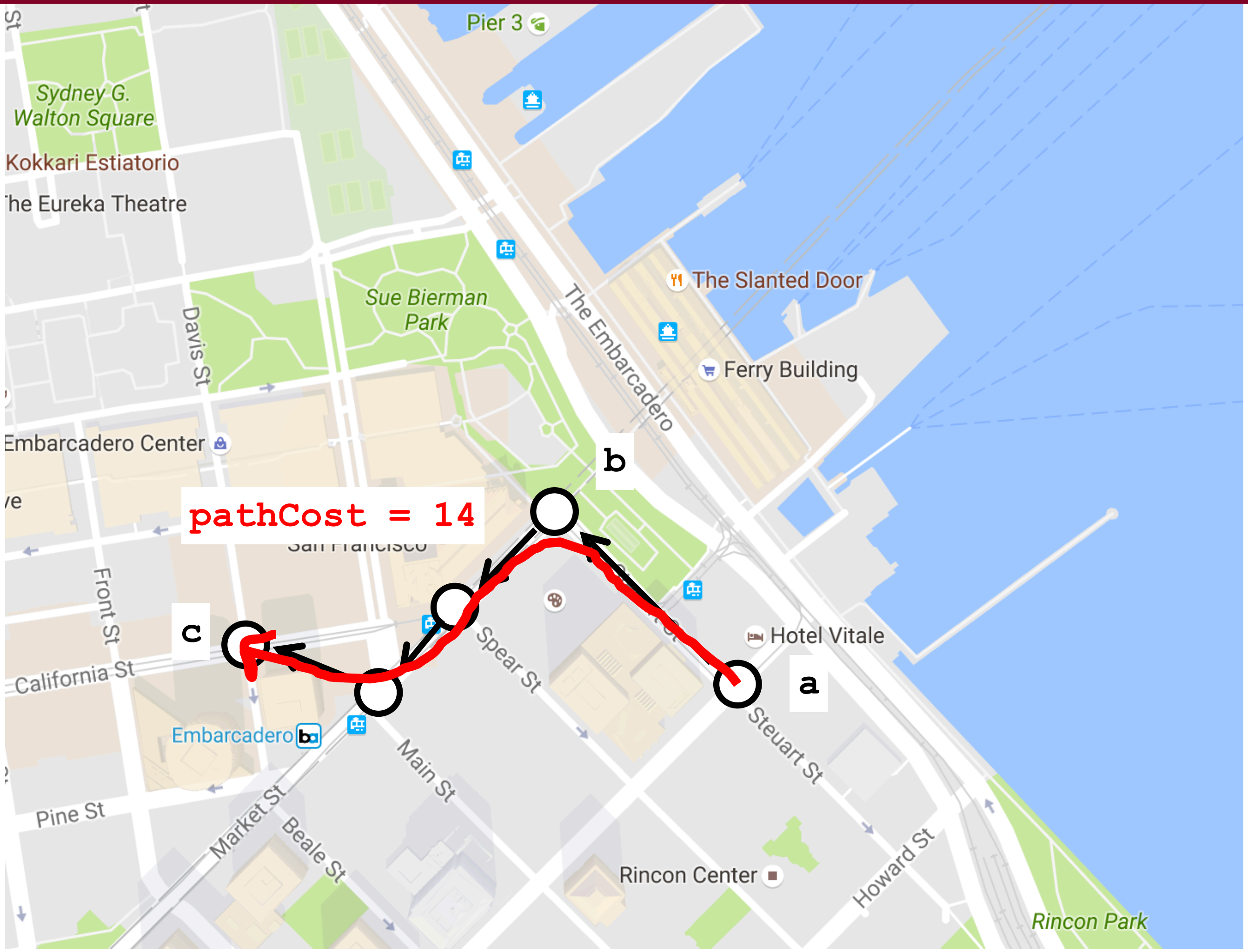
Road Map Edge



Road Map Edge Cost



Road Map Path Cost



Could Google Just Precompute?

How many nodes in google maps graph?

~ 75 million

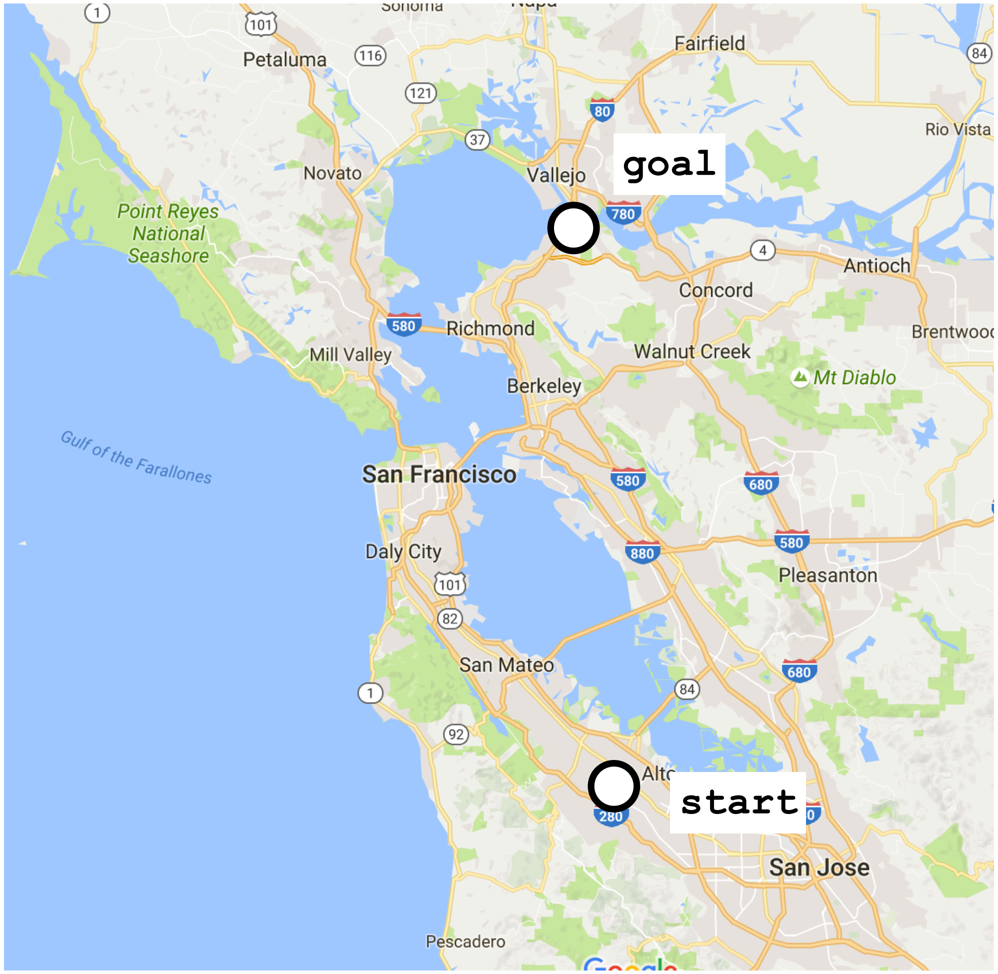
n^2

6×10^{15}

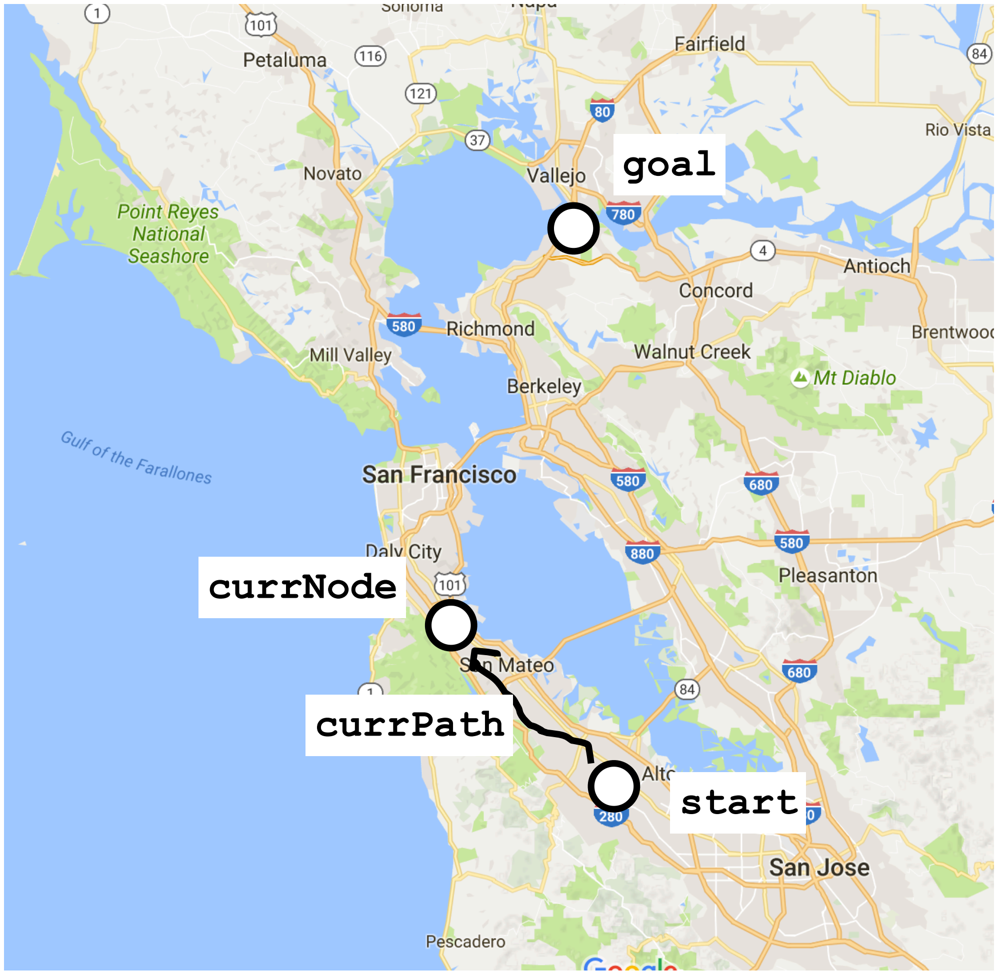
1 petasecond = 31.7 million years

Can you think of a heuristic?

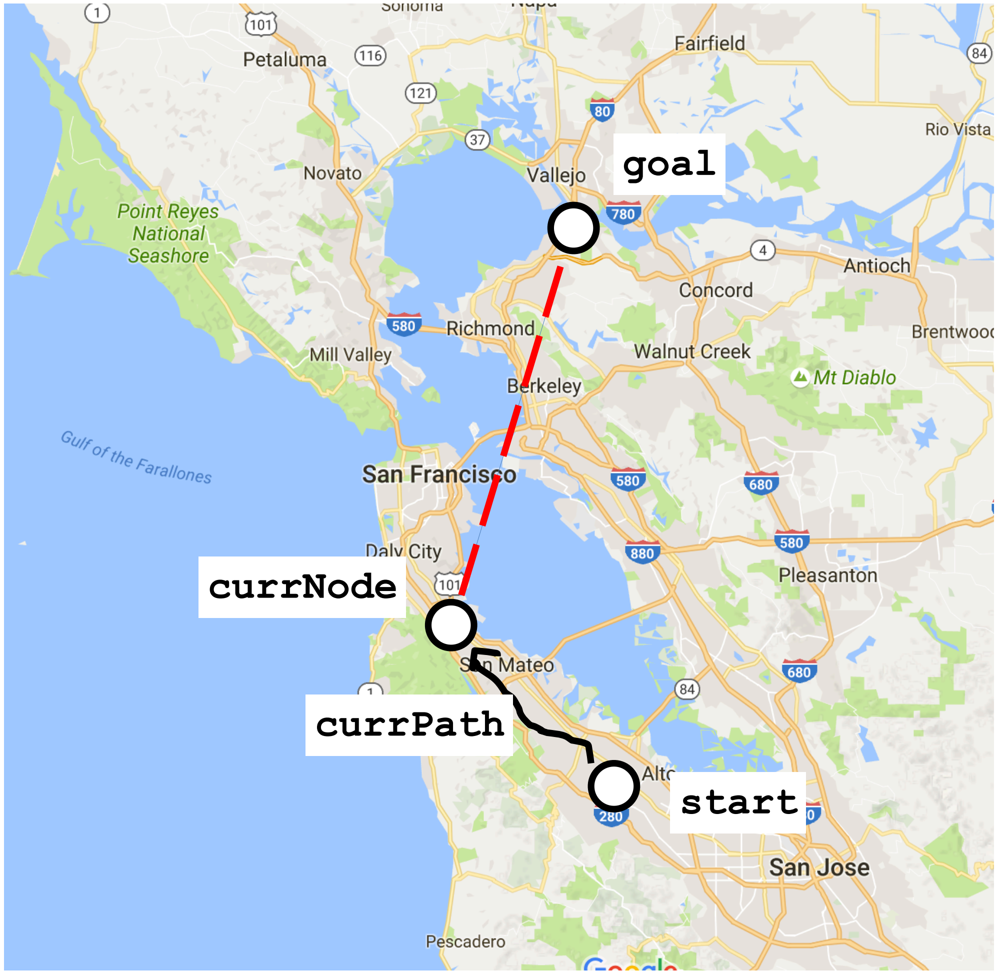
Road Map Heuristic



Road Map Heuristic

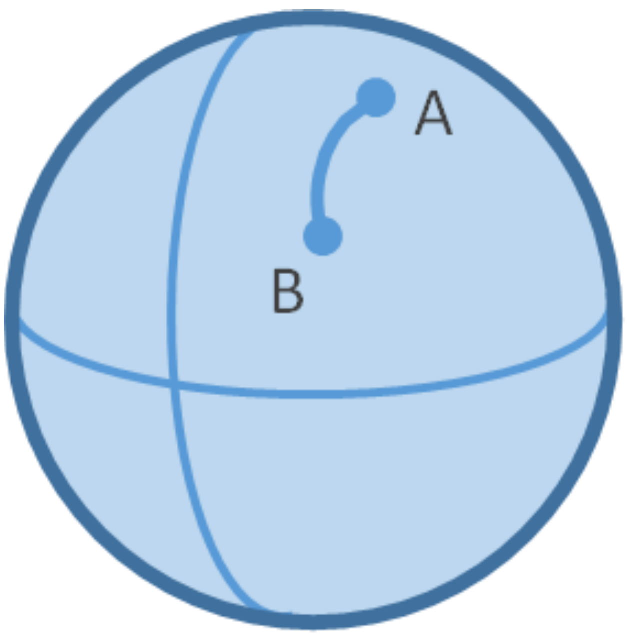


We must *underestimate* this time



Direct Highway

$$\text{Heuristic} = \frac{\text{Distance on surface of earth}}{\text{Speed on fastest highway}}$$

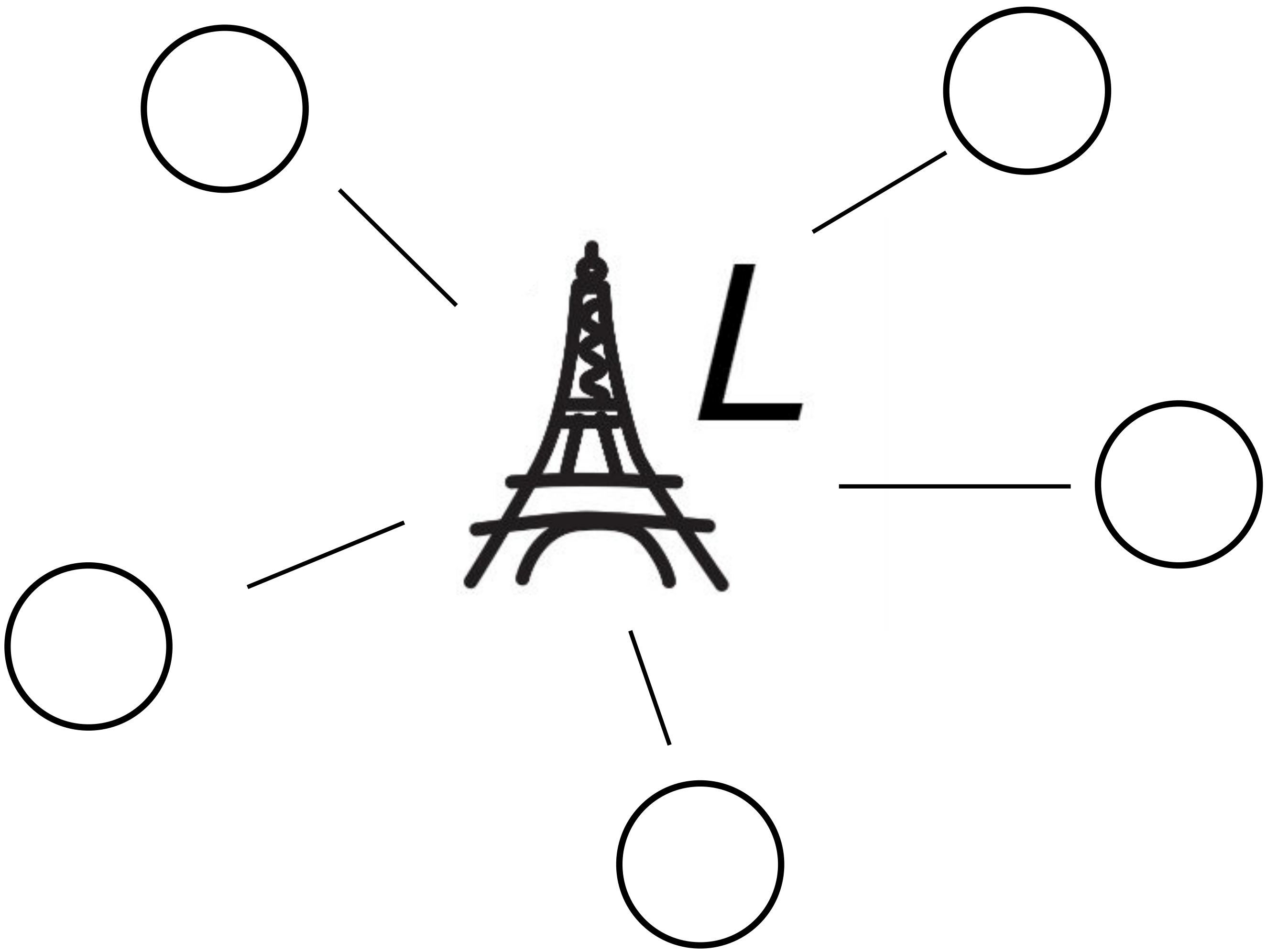


For Trailblazer:

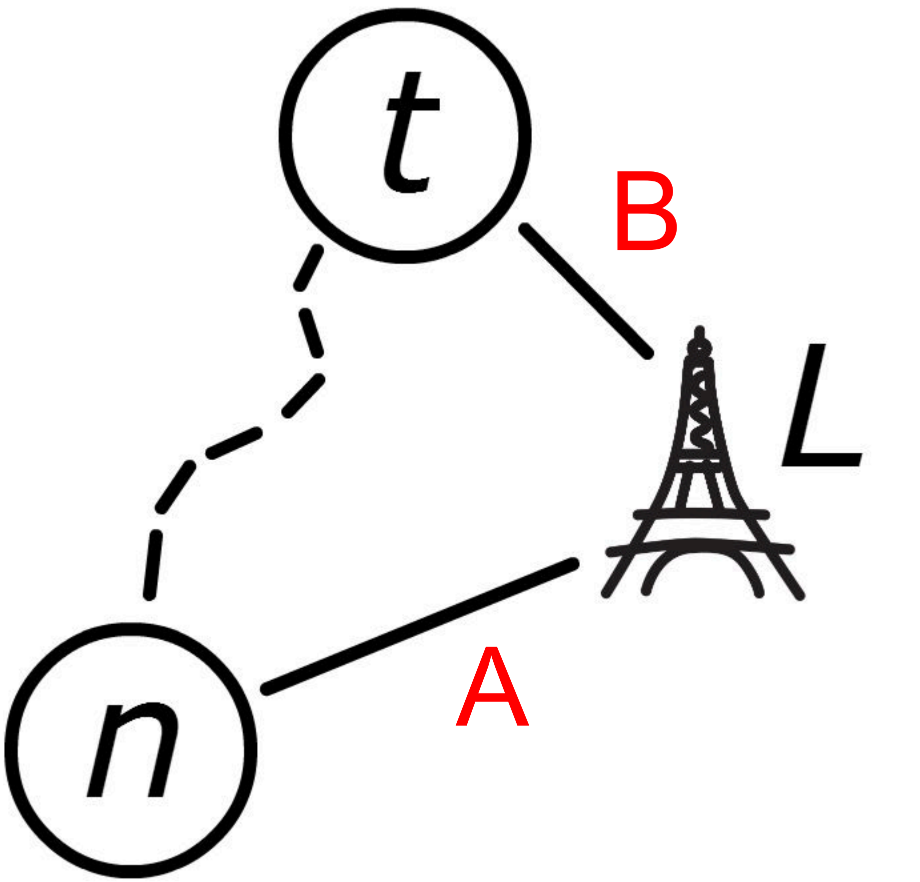
Distance on surface of earth is **`getCrowFlyDistance()`**

Speed on fastest highway is **`getMaxRoadSpeed()`**

Distance to Landmarks



Landmark Heuristic



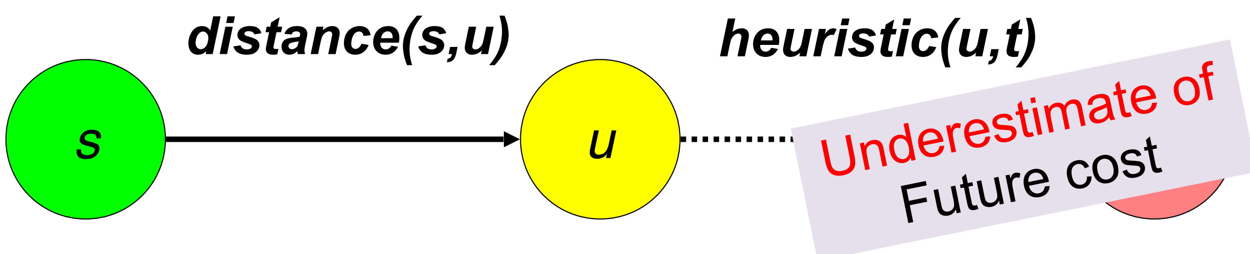
Distance < abs(A – B)

Best of All Heuristics

$$h = \max(h_1, h_2, ..., h_n)$$

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)

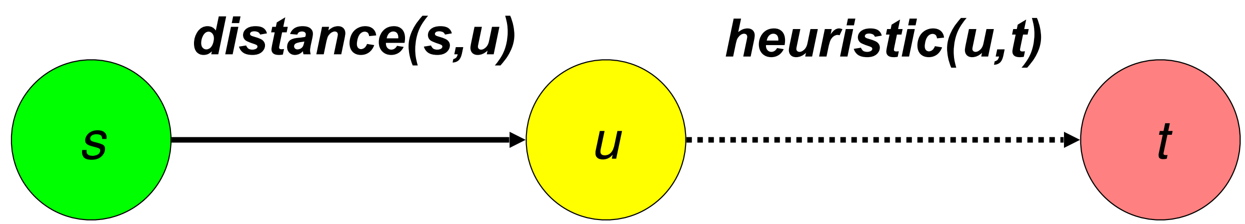


We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



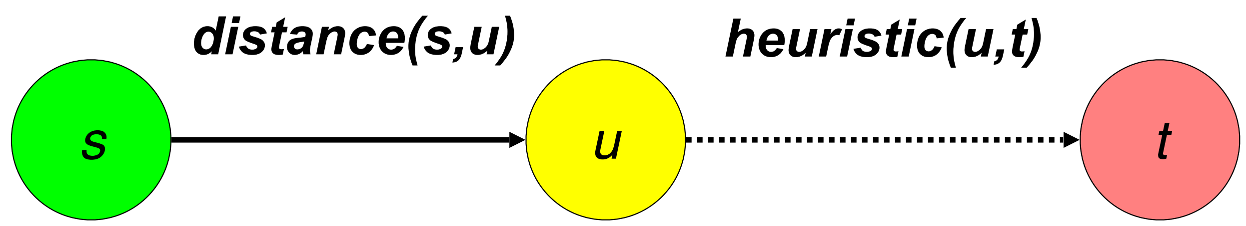
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Same as Dijkstra

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



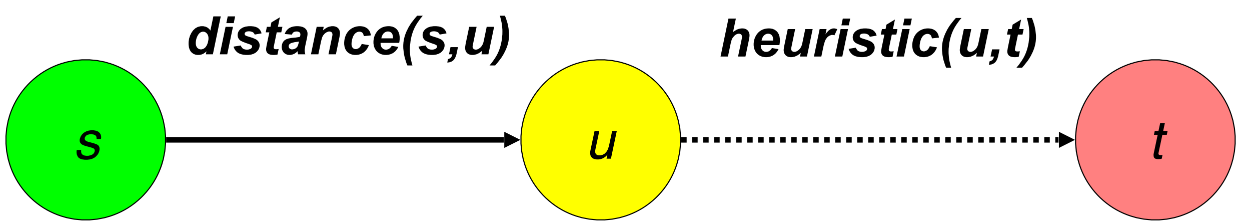
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Will be the same or faster than Dijkstra, and will find the shortest path (this is the only "admissible" heuristic for A*.

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



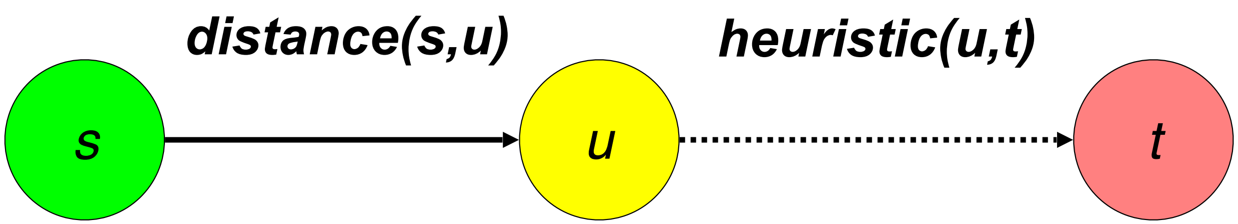
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Will only follow the best path, and will find the best path fastest (but requires perfect knowledge)

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Won't necessarily find shortest path (but might run even faster)

Admissible Heuristic

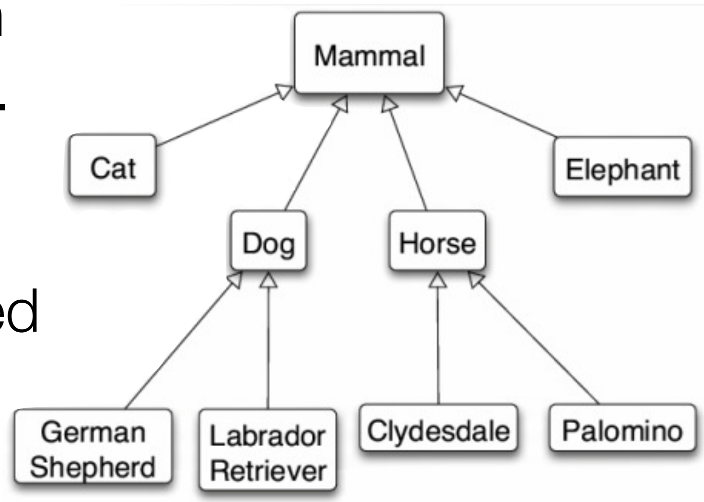
Definition: An admissible heuristic always underestimates the true cost.

Could you precompute this for all your vertices? Yes, but it would not be feasible.

Inheritance in C++

inheritance: A way to form new classes based on existing classes, taking on their attributes/behavior.

- a way to indicate that classes are related
- a way to share code between two or more related classes (a **hierarchy**)



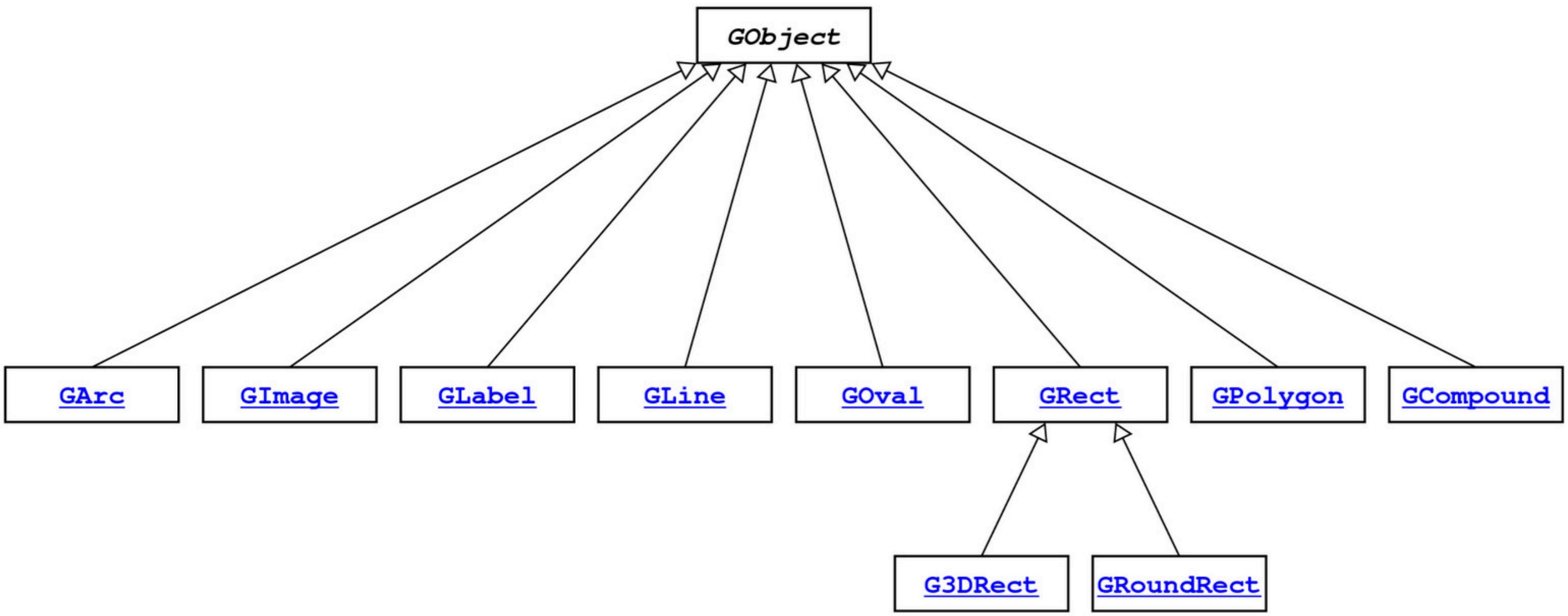
One class can *extend* another, absorbing its data/behavior.

- **superclass** (base class): Parent class that is being extended.
- **subclass** (derived class): Child class that inherits from the superclass.
  - Subclass gets a copy of every field and method from superclass.
  - Subclass can add its own behavior, and/or change inherited behavior.

GObject Hierarchy

The Stanford C++ library contains a hierarchy of graphical objects based on a common base class named `GObject`.

- `GArc, GCompound, GImage, GLabel, GLine, GOval, GPolygon, GRect, G3DRect, GRoundRect, ...`



GObject Members

`GObject` defines the state and behavior common to all shapes:

- `contains(x, y)`
- `getColor(), setColor(color)`
- `getHeight(), getWidth(), getLocation(), setLocation(x, y)`
- `getX(), getY(), setX(x), setY(y), move(dx, dy)`
- `setVisible(visible)`
- `toString()`

The subclasses add state and behavior unique to them:

`GLabel:`
- `get/setFont`
- `get/setLabel`
-
- `...`

`GLine:`
- `get/setStartPoint`
- `get/setEndPoint`
-
- `...`

`GPolygon:`
- `addEdge`
- `addVertex`
- `get/setFillColor`
- `...`

Example: Employees

Imagine a company with the following **employee regulations**:

- All employees work 40 hours / week.
- Employees make $40,000 per year plus $500 for each year worked,
  - except for lawyers who get twice the usual pay,
    and programmers who get the same $40k base but $2000 for each year worked.
- Employees have 2 weeks of paid vacation days per year,
  - except for programmers who get an extra week (a total of 3).
- Employees should use a yellow form to apply for leave,
  - except for programmers who use a pink form.

Each type of employee has some unique behavior:

- **Lawyers** know how to sue.
- **Programmers** know how to write code.
- **Secretaries** know how to take dictation.
- **Legal Secretaries** know how to take dictation and how to file legal briefs.

Employee Class

```
// Employee.h
class Employee {
public:
    Employee(string name, int years);
    virtual int hours() const;
    virtual string name() const;
    virtual double salary() const;
    virtual int vacationDays() const;
    virtual string vacationForm() const;
    virtual int years() const;

private:
    string myName;
    int myYears;
};
```

```
// Employee.cpp
Employee::Employee(string name, int years) {
    myName = name;
    myYears = years;
}

int Employee::hours() const {
    return 40;
}

string Employee::name() const {
    return myName;
}

double Employee::salary() const {
    return 40000.0 + (500 * myYears);
}

int Employee::vacationDays() const {
    return 10;
}

string Employee::vacationForm() const {
    return "yellow";
}

int Employee::years() const {
    return myYears;
}
```

Exercise: Employees

Exercise: Implement classes Lawyer and Programmer.

Lawyer

- A Lawyer remembers what **law school** he/she went to.
- Lawyers make twice as much **salary** as normal employees.
- Lawyers know how to **sue** people (unique behavior).

Programmer

- Programmers make the same base salary as normal employees, but they earn a **bonus of $2k/year** instead of $500/year.
- Programmers fill out the **pink form** rather than yellow for vacations.
- Programmers get **3 weeks of vacation** rather than 2.
- Programmers know how to write **code** (unique behavior).

Overriding

- **override**: To replace a superclass's member function by writing a new version of that function in a subclass.
- **virtual function**: One that is allowed to be overridden.
  - Must be declared with `virtual` keyword in superclass.

```
// Employee.h
virtual string vacationForm();


// Employee.cpp
string Employee::vacationForm() {
    return "yellow";
}
```

```
// Programmer.h
virtual string vacationForm();


// Programmer.cpp
string Programmer::vacationForm() {
    return "pink";    // override!
}
```

If you "override" a non-virtual function, it actually just puts a second copy of that function in the subclass, which can be confusing later.

* Virtual has some subtleties. For example, destructors in inheritance hierarchies should always be declared virtual or else memory may not get cleaned up properly; ugh.

Calling the Superclass Constructor

```
SubclassName::SubclassName(params) : SuperclassName(params) {
    statements;
}
```

To call a superclass constructor from subclass constructor, use an *initialization list*, with a colon after the constructor declaration.

Example:

```
Lawyer::Lawyer(string name, string lawSchool, int years) :
          Employee(name, years) {
    // calls Employee constructor first
    mylawSchool = lawSchool;
}
```

Calling the Superclass Member

`SuperclassName::memberName(params)`

To call a superclass overridden member from subclass member.

Example:

```
double Lawyer::salary() { // paid twice as much
    return Employee::salary() * 2;
}
```

Notes:

- Subclass cannot access private members of the superclass.
- You only need to use this syntax when the superclass's member has been overridden.
- If you just want to call one member from another, even if that member came from the superclass, you don't need to write `Superclass::` .

Lawyer.h

```cpp
#pragma once

#include "Employee.h"
#include <string>

class Lawyer : public Employee {
    // I now have an hours, name, salary, etc. method. yay!
public:
    Lawyer(string name, string lawSchool, int years);
    virtual double salary() const;
    void sue(string person);

private:
    string myLawSchool;
};
```

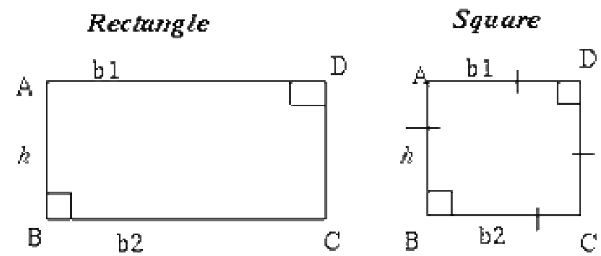
Lawyer.cpp

```
#include "Lawyer.h"

// call the constructor of Employee superclass?
Lawyer::Lawyer(string name, string lawSchool, int years)
: Employee(name, years) {
    myLawSchool = lawSchool;
}

// overriding: replace version from Employee class
double Lawyer::salary() const {
    return Employee::salary() * 2;
}

void Lawyer::sue(string person) {
    cout << "See you in court, " << person << endl;
}
```

Perils of Inheritance (i.e., think before you inherit!)

Consider the following places you might use inheritance:

- class `Point3D` extends `Point2D` and adds z-coordinate
- class `Square` extends `Rectangle` (or vice versa?)
- class `SortedVector` extends `Vector`, keeps it in sorted order



What's wrong with these examples? Is inheritance good here?

- `Point2D`'s `distance()` function is wrong for 3D points
- `Rectangle` supports operations a `Square` shouldn't (e.g. `setWidth`)
- `SortedVector` might confuse client; they call insert at an index, then check that index, and the element they inserted is elsewhere!

Private Inheritance

```
class Name : private SuperclassName { ...
```

private inheritance: Copies code from superclass but does not publicly advertise that your class extends that superclass.

- Good for cases where you want to inherit another class's code, but you don't want outside clients to be able to randomly call it.
- Example: Have `Point3D` privately extend `Point2D` and add z-coordinate functionality.
- Example: Have `SortedVector` privately extend `Vector` and add only the public members it feels are appropriate (e.g., no `insert`).

Pure Virtual Functions

virtual **returntype name(params)** = 0;

pure virtual function: Declared in superclass's .h file and set to 0 (null). An absent function that has not been implemented.

- Must be implemented by any subclass, or it cannot be used.
- A way of forcing subclasses to add certain important behavior.

```
class Employee {
    ...
    virtual void work() = 0;  // every employee does
                              // some kind of work
};
```

FYI: In Java, this is called an *abstract method*.

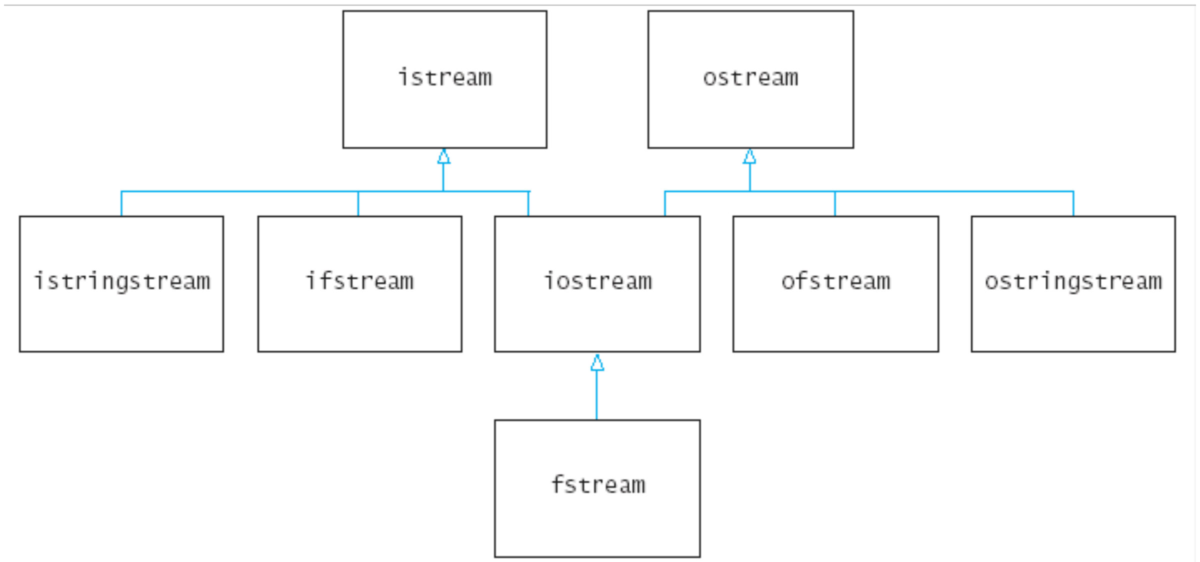
Multiple Inheritance

class ***Name*** : public ***Superclass1***, public ***Superclass2***, ...

multiple inheritance: When one subclass has multiple superclasses.

- Forbidden in many OO languages (e.g. Java) but allowed in C++.
- Convenient because it allows code sharing from multiple sources.
- Can be confusing or buggy, e.g. when both superclasses define a member with the same name.

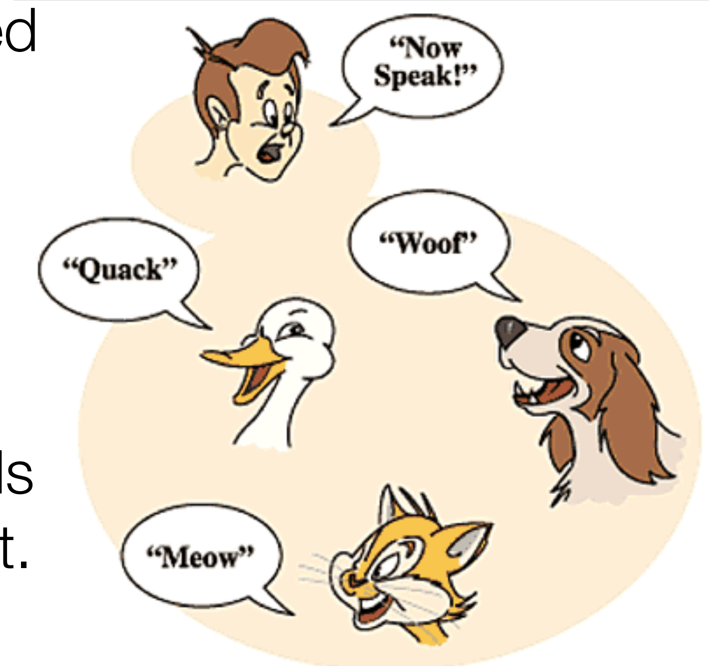
Example: The C++ I/O streams use multiple inheritance:



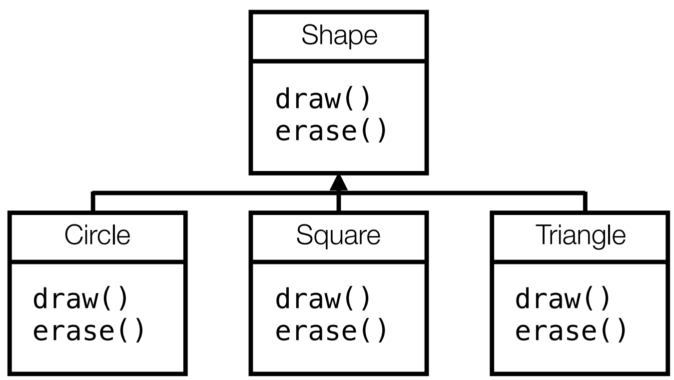
Polymorphism

polymorphism: Ability for the same code to be used with different types of objects and behave differently with each.

- Templates provide *compile-time* polymorphism. Inheritance provides *run-time* polymorphism.



Idea: Client code can call a method on different kinds of objects, and the resulting behavior will be different.



Polymorphism and Pointers

A pointer of type *T* can point to any subclass of *T*.

```
Employee* edna = new Lawyer("Edna", "Harvard", 5);
Secretary* steve = new LegalSecretary("Steve", 2);
World* world     = new WorldMap("map-stanford.txt");
```

When a member function is called on edna, it behaves as a Lawyer.

- (This is because the employee functions are declared virtual.)
- You can *not* call any `Lawyer`-only members on `edna` (e.g. `sue`).
  You can *not* call any `LegalSecretary`-only members on `steve` (e.g. `fileLegalBriefs`).

Polymorphism Example

You can use the object's extra functionality by casting.

```
Employee* edna = new Lawyer("Edna", "Harvard", 5);
edna->vacationDays();                              // ok
edna->sue("Stuart");                               // compiler error
((Lawyer*) edna)->sue("Stuart");                   // ok
```

You should not cast a pointer to something that it is not.

- It will compile, but the code will crash (or behave unpredictably) when you try to run it

```
Employee* paul = new Programmer("Paul", 3);
paul->code();                                            // compiler
error
((Programmer*) paul)->code();                            // ok
((Lawyer*) paul)->sue("Marty");                          // crash!
```

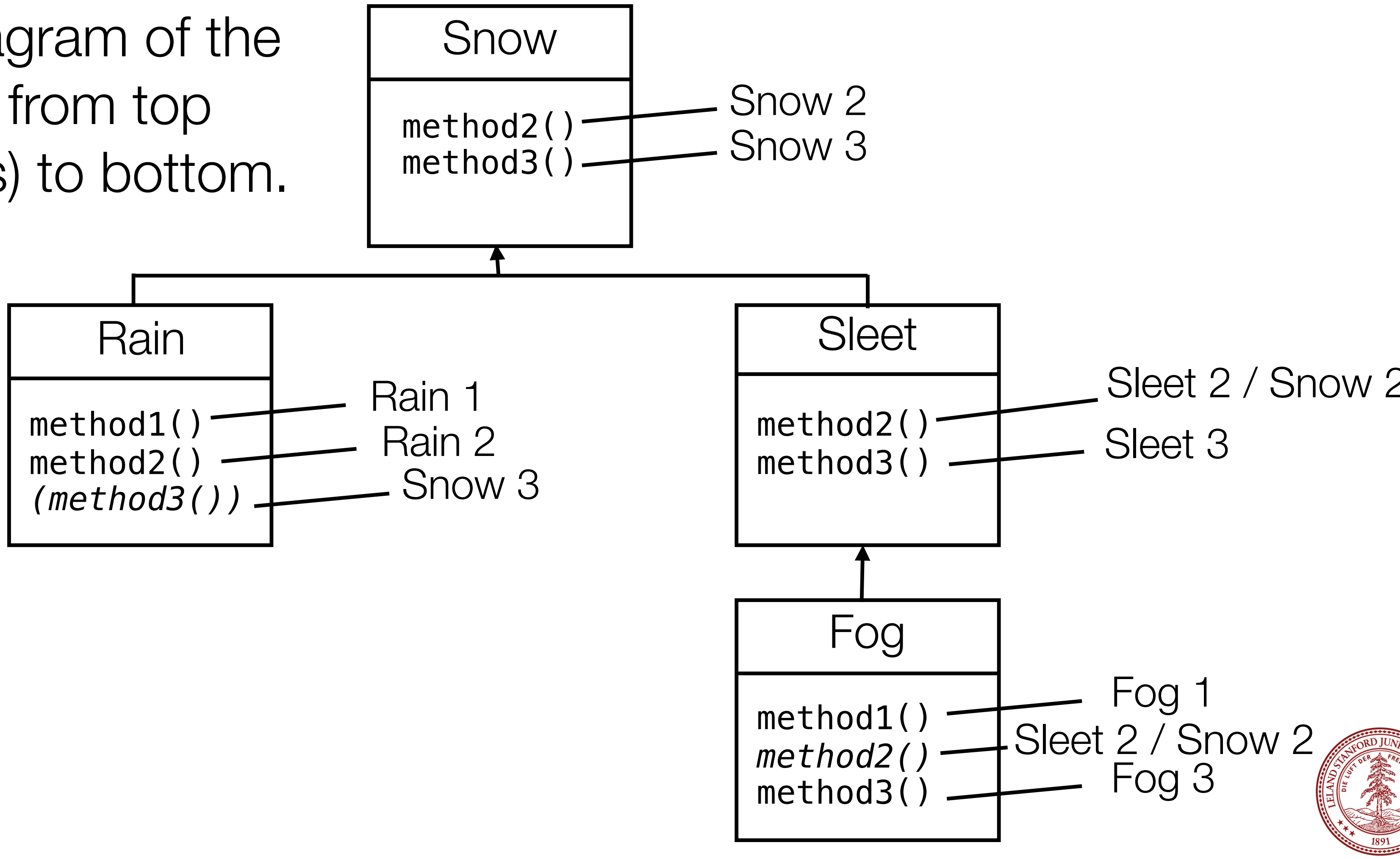
Polymorphism Mystery

```cpp
class Snow {
public:
    virtual void method2() {
        cout << "Snow 2" << endl;
    }
    virtual void method3() {
        cout << "Snow 3" << endl;
    }
};

class Rain : public Snow {
public:
    virtual void method1() {
        cout << "Rain 1" << endl;
    }
    virtual void method2() {
        cout << "Rain 2" << endl;
    }
};
```

```cpp
class Sleet : public Snow {
public:
    virtual void method2() {
        cout << "Sleet 2" << endl;
        Snow::method2();
    }
    virtual void method3() {
        cout << "Sleet 3" << endl;
    }
};

class Fog : public Sleet {
public:
    virtual void method1()   {
        cout << "Fog 1" <<   endl;
    }
    virtual void method3()   {
        cout << "Fog 3" <<   endl;
    }
};
```

Diagramming classes

Draw a diagram of the classes from top (superclass) to bottom.



Mystery Problem

```
Snow* var1 = new Sleet();
var1->method2(); // What's the output?
```

To find the behavior/output of calls like the one above:

1. Look at the variable's type.
   If that type does not have that member: COMPILER ERROR.
2. Execute the member.
   Since the member is virtual: behave like the object's type, not like the variable's type.

Example 1

Q: What is the result of the following call?

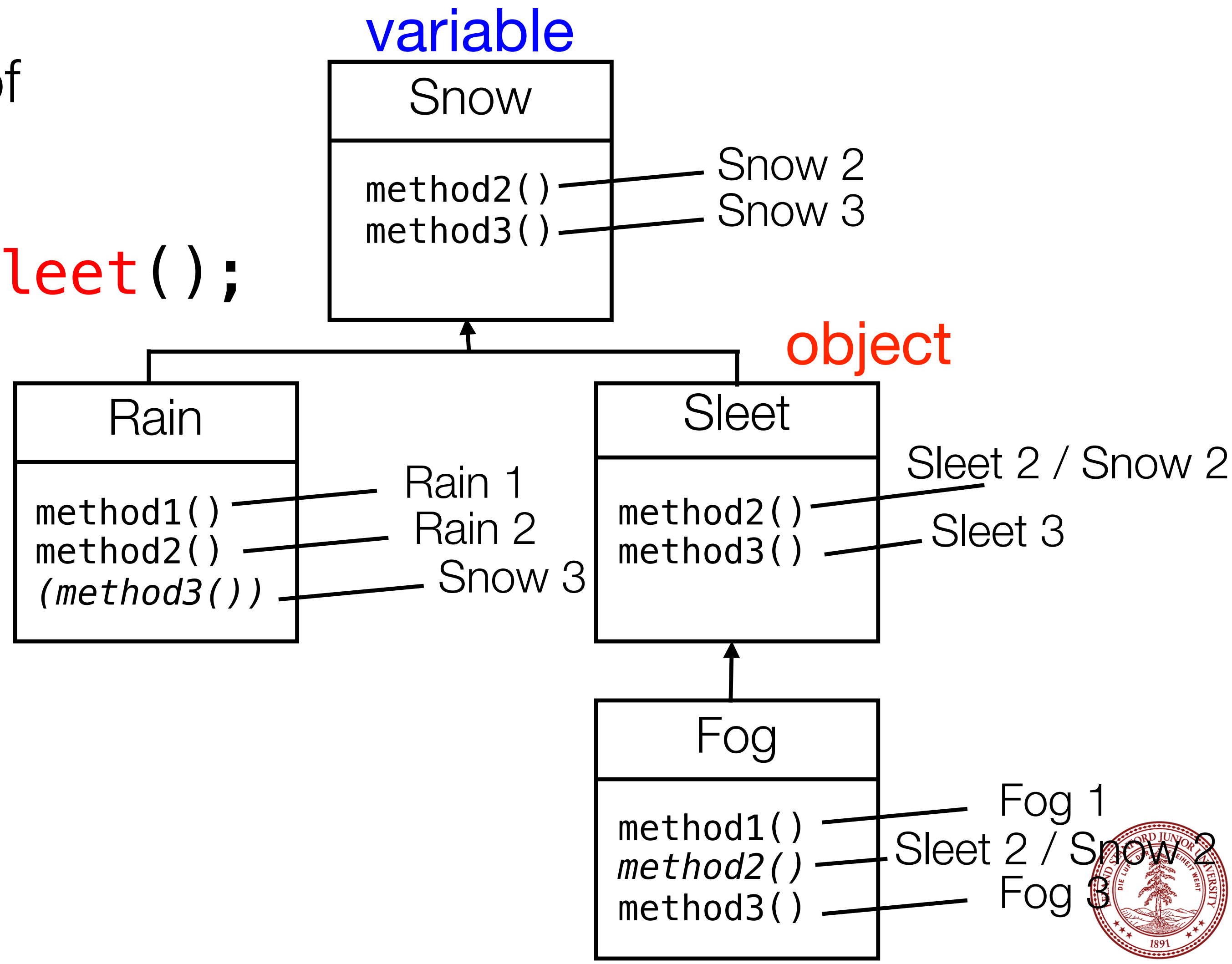
```
Snow* var1 = new Sleet();
var1->method2();
```

A. Snow 2

B. Rain 2

C. Sleet 2
   Snow 2

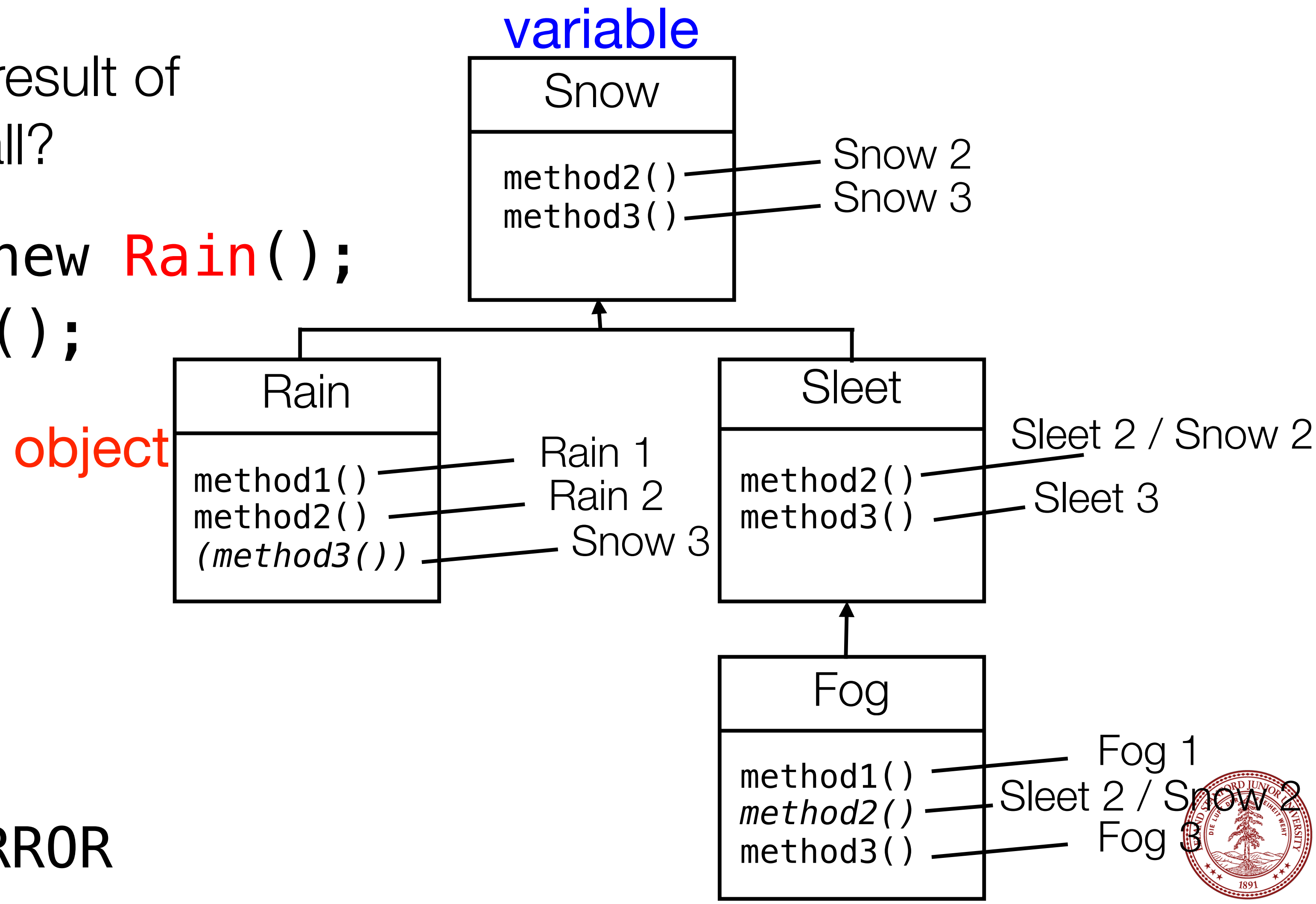
D. COMPILER ERROR



Example 2

Q: What is the result of the following call?

```
Snow* var2 = new Rain();
var2->method2();
```



A. Snow 1

B. Rain 1

C. Snow 1
Rain 1

D. COMPILER ERROR

Example 3

Q: What is the result of the following call?

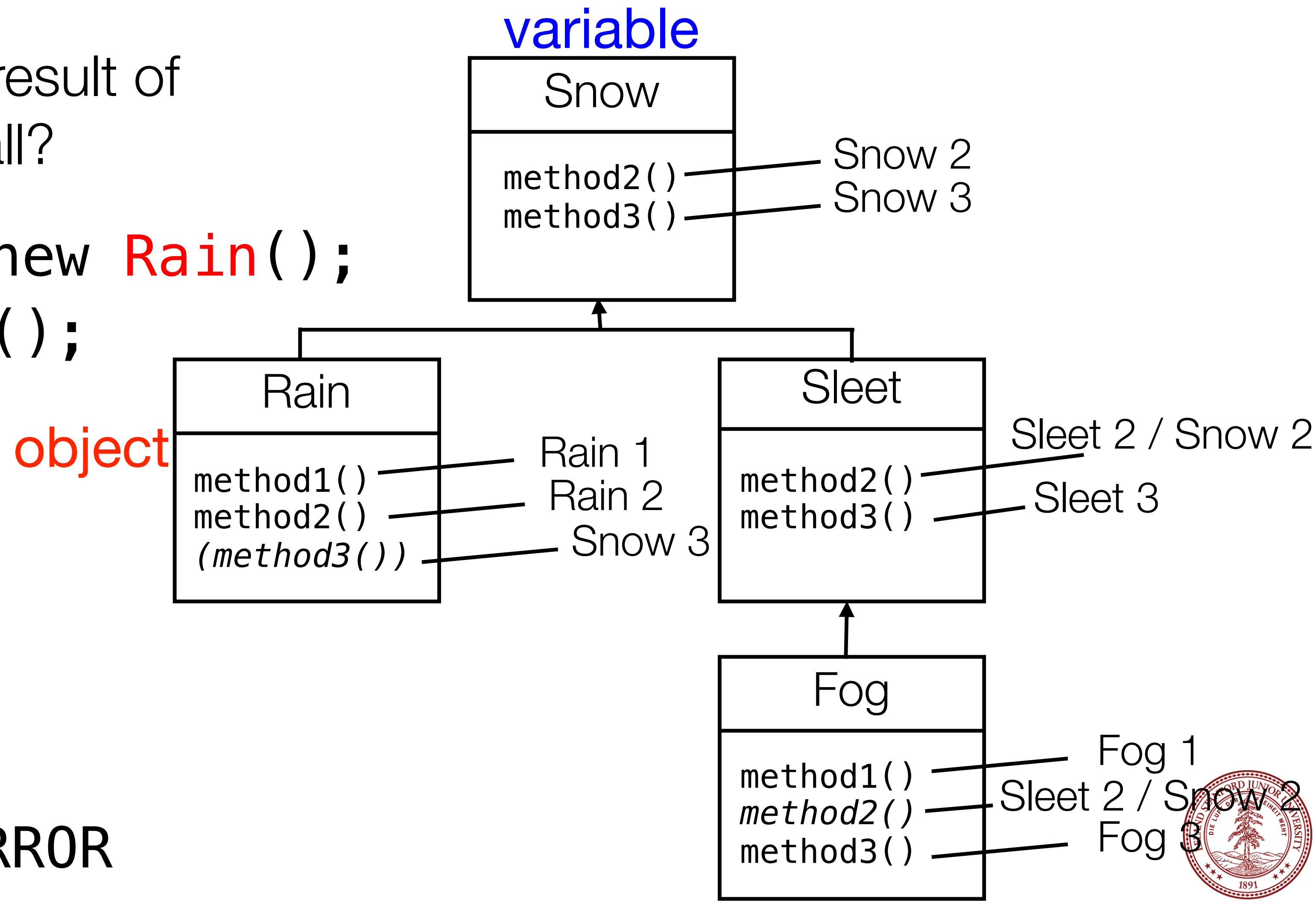
```
Snow* var3 = new Rain();
var3->method2();
```

A. Snow 2

B. Rain 2

C. Sleet 2
Snow 2

D. COMPILER ERROR



Mystery with type cast

```
Snow* var4 = new Rain();
((Sleet *) var4->method2(); // What's the output?
```

If the mystery problem has a type cast, then:

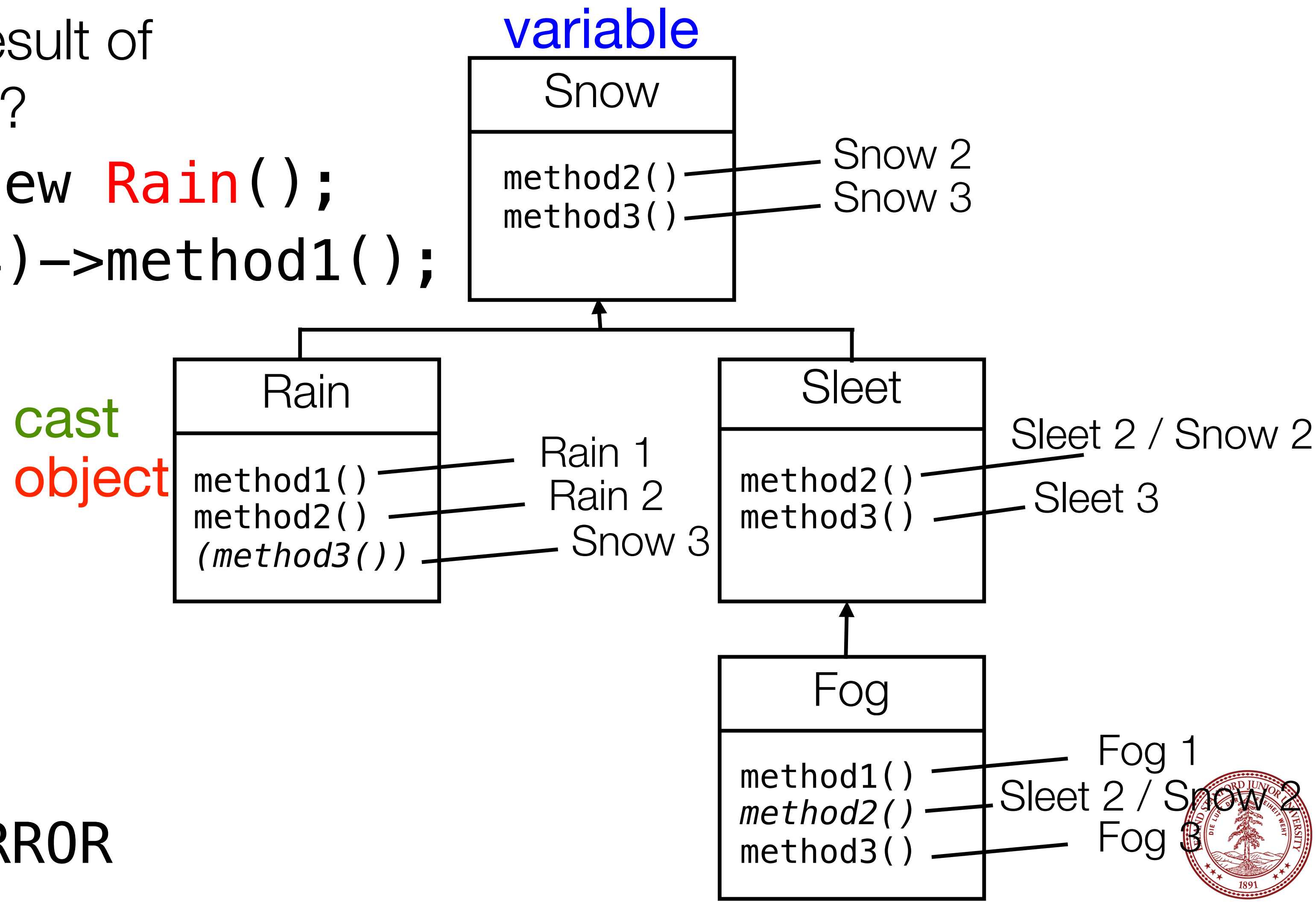
1. Look at the cast type.
   If that type does not have the method: COMPILER ERROR.

   (Note: if the object's type was not equal to or a subclass of the cast type, the code would CRASH / have unpredictable behavior.)
2. Execute the member.
   Since the member is virtual: behave like the object's type, not like the variable's type.

Example 4

Q: What is the result of the following call?

```
Snow* var4 = new Rain();
((Rain *) var4)->method1();
```



A. Snow 1

B. Rain 1

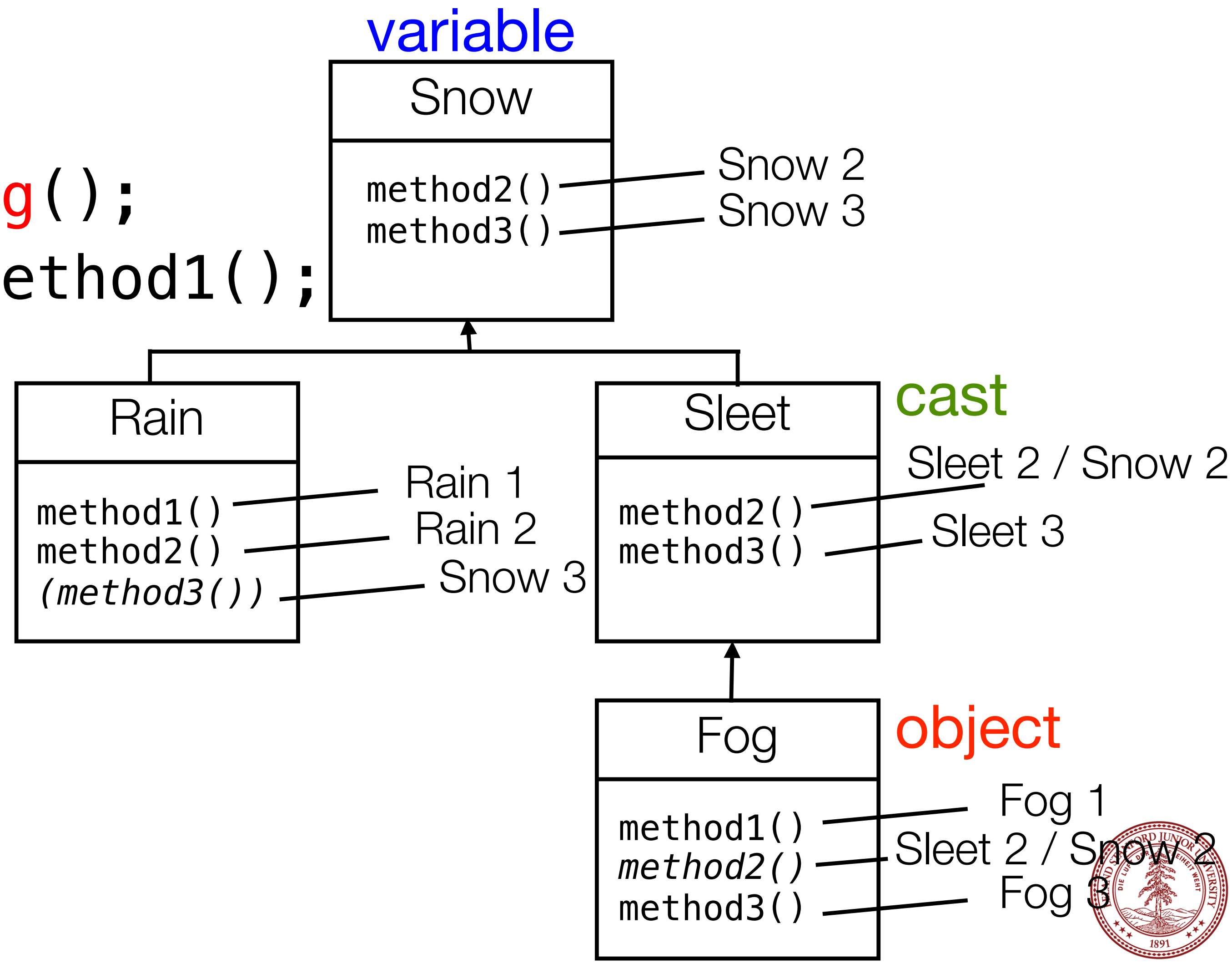
C. Sleet 1

D. COMPILER ERROR

Example 5

Q: What is the result of the following call?

```
Snow* var5 = new Fog();
((Sleet *) var5)->method1();
```



A. Snow 1

B. Sleet 1

C. Fog 1

D. COMPILER ERROR

Example 6

Suppose we add the following method to base class Snow:

```
virtual void method4() {
  cout << "Snow 4" << endl;
  method2();
}
```

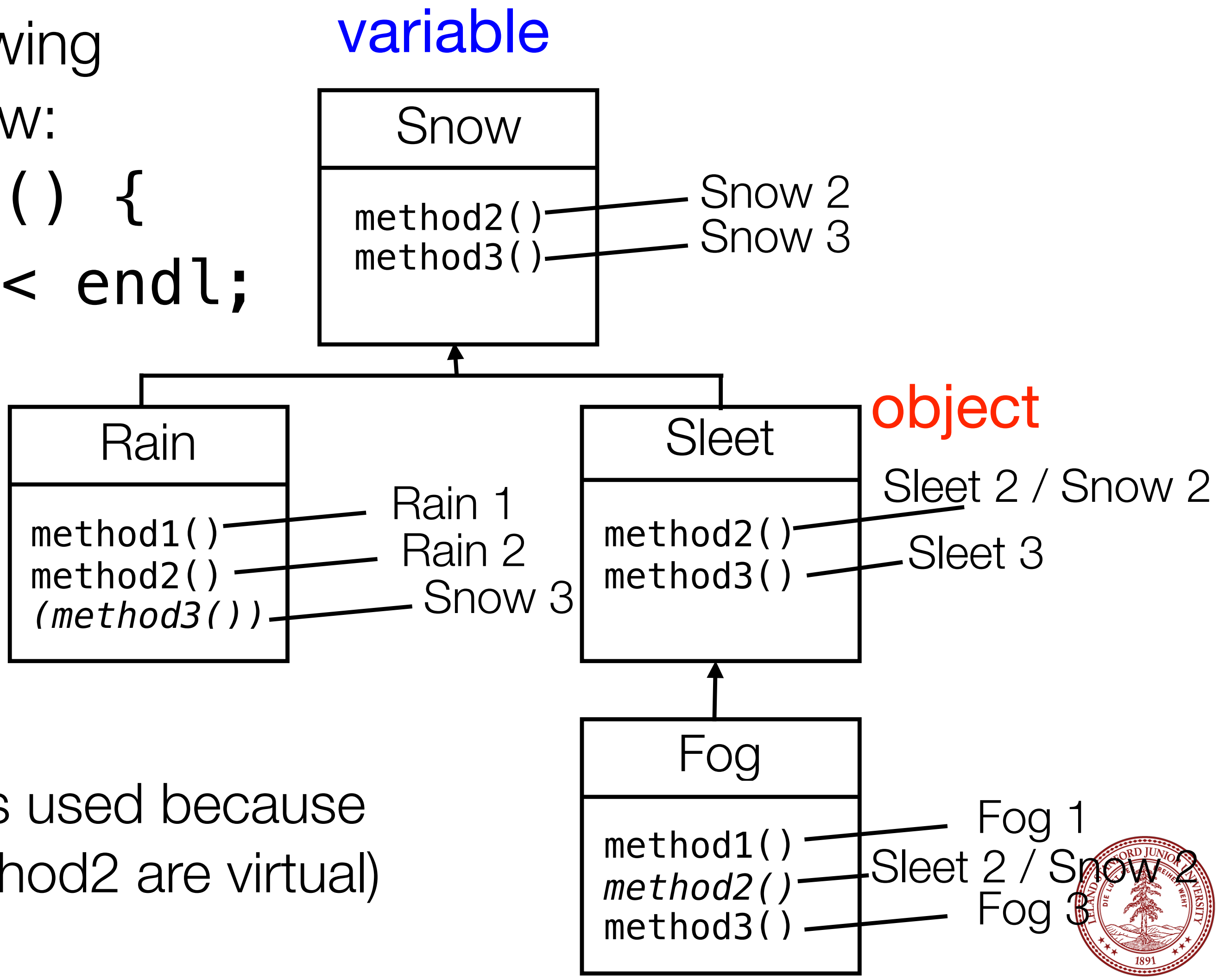
What is the output?

```
Snow* var8 = new Sleet();
var8->method4();
```

Answer:

```
Snow 4
Sleet 2
Snow 2
```

(Sleet's method2 is used because method 4 and method2 are virtual)



Example 7

What is the output of the following call?

```
Snow* var6 = new Sleet();
((Rain*) var6)->method1();
```

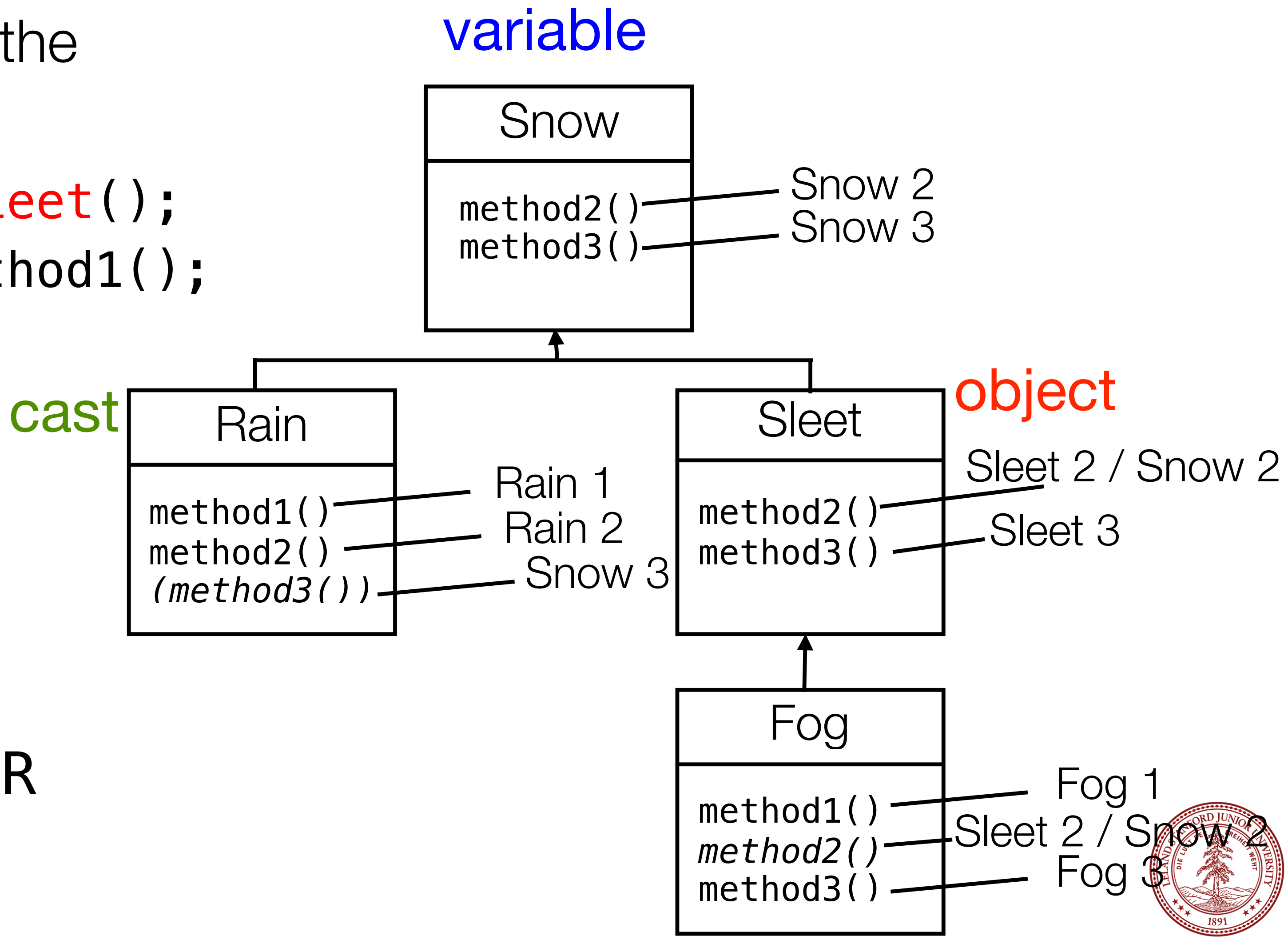
A. Snow 1

B. Sleet 1

C. Fog 1

D. COMPILER ERROR

E. CRASH



References and Advanced Reading

- **References:**
  - C++ Inheritance: https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm
  - C++ Polymorphism: https://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm

- **Advanced Reading:**
  - http://stackoverflow.com/questions/5854581/polymorphism-in-c
  - https://www.codingunit.com/cplusplus-tutorial-polymorphism-and-abstract-base-class

Extra Slides