Object Oriented Programming & Inheritance

Nick Sumner
wsumner@sfu.ca
Why care about Object Oriented Programming?

- Superficial reasons are straightforward:
Why care about Object Oriented Programming?

- Superficial reasons are straightforward:
  - Much of the world’s code is written in OO languages
  - Employers consider OOP a core introductory skill
  - Chances are you will need to understand it to get a job
Why care about Object Oriented Programming?

- Superficial reasons are straightforward:
  - Much of the world’s code is written in OO languages
  - Employers consider OOP a core introductory skill
  - Chances are you will need to understand it to get a job

- But there are detractors
  - “OOP is inefficient”
  - “OOP makes it hard to share data easily”
Why care about Object Oriented Programming?

• Superficial reasons are straightforward:
  – Much of the world’s code is written in OO languages
  – Employers consider OOP a core introductory skill
  – Chances are you will need to understand it to get a job

• But there are detractors
  – “OOP is inefficient”
  – “OOP makes it hard to share data easily”

• Applied well and thoughtfully, it helps solve real problems
  – Like any tool, if you apply it poorly, it won’t work well
  – If you apply it universally or dogmatically, you will miss out on better tools
  – You need to know how to use a tool to get value out of it
Why care about Object Oriented Programming?

- Superficial reasons are straightforward:
  - Much of the world’s code is written in OO languages
  - Employers consider OOP a core introductory skill
  - Chances are you will need to understand it to get a job

- But there are detractors
  - “OOP is inefficient”
  - “OOP makes it hard to share data easily”

- Applied well and thoughtfully, it helps solve real problems
  - Like any tool, if you apply it poorly, it won’t work well
  - If you apply it universally or dogmatically, you will miss out on better tools
  - You need to know how to use a tool to get value out of it

- OOP will not solve your design for you, but it can be an effective tool
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
  - Most employers will expect you to have seen it from year 1
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
  - Most employers will expect you to have seen it from year 1
  - You will be competing on the job market with students doing it from year 1
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
  - Most employers will expect you to have seen it from year 1
  - You will be competing on the job market with students doing it from year 1

- But many schools teach it very badly
  - OOP textbooks were notoriously bad in the early 2000s
  - Many were written by people who did not know what they were doing
  - Many faculty did not learn it well themselves
Our Goal

● I will assume you have basic, introductory, OOP experience
  – Most schools teach this in year 1 (ours does a little & is aiming to get better)
  – Most employers will expect you to have seen it from year 1
  – You will be competing on the job market with students doing it from year 1

● But many schools teach it very badly
  – OOP textbooks were notoriously bad in the early 2000s
  – Many were written by people who did not know what they were doing
  – Many faculty did not learn it well themselves
  – This is one of the reasons people complain about OOP
Our Goal

I will assume you have basic, introductory, OOP experience
- Most schools teach this in year 1 (ours does a little & is aiming to get better)
- Most employers will expect you to have seen it from year 1
- You will be competing on the job market with students doing it from year 1

But many schools teach it very badly
- OOP textbooks were notoriously bad in the early 2000s
- Many were written by people who did not know what they were doing
- Many faculty did not learn it well themselves
- This is one of the reasons people complain about OOP

Our goal with OOP is to make you better than that
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
  - Most employers will expect you to have seen it from year 1
  - You will be competing on the job market with students doing it from year 1

- But many schools teach it very badly
  - OOP textbooks were notoriously bad in the early 2000s
  - Many were written by people who did not know what they were doing
  - Many faculty did not learn it well themselves
  - This is one of the reasons people complain about OOP

- Our goal with OOP is to make you better than that
  - Regardless of the language you work in, I recommend: *Effective Java, C++ Coding Standards, Practical Object-Oriented Design in Ruby*
Our Goal

- I will assume you have basic, introductory, OOP experience
  - Most schools teach this in year 1 (ours does a little & is aiming to get better)
  - Most employers will expect you to have seen it from year 1
  - You will be competing on the job market with students doing it from year 1

- But many schools teach it very badly
  - OOP textbooks were notoriously bad in the early 2000s
  - Many were written by people who did not know what they were doing
  - Many faculty did not learn it well themselves
  - This is one of the reasons people complain about OOP

- Our goal with OOP is to make you better than that
  - Regardless of the language you work in, I recommend:
    - *Effective Java*, *C++ Coding Standards*, *Practical Object-Oriented Design in Ruby*

  Treat these as guides rather than laws. Dogma has no value. Understand the cost/benefit.
What is OOP?

- This is a matter of more debate than expected
What is OOP?

- This is a matter of more debate than expected
- **Classically:**
  - A combination of data and code
  - Abstraction, Encapsulation, Inheritance, Polymorphism
What is OOP?

- This is a matter of more debate than expected

Classically:
- A combination of data and code
- Abstraction, Encapsulation, Inheritance, Polymorphism
- “An object is a value exporting a procedural interface to data or behavior” [Cook 2009]
What is OOP?

- This is a matter of more debate than expected
- **Classically:**
  - A combination of data and code
  - Abstraction, Encapsulation, Inheritance, Polymorphism
  - "An object is a value exporting a procedural interface to data or behavior" [Cook 2009]
  - Objects are “sites of higher level behaviors more appropriate for use as dynamic components” [Kay 1993]
What is OOP?

- This is a matter of more debate than expected

- Classically:
  - A combination of data and code
  - Abstraction, Encapsulation, Inheritance, Polymorphism
  - “An object is a value exporting a procedural interface to data or behavior” [Cook 2009]
  - Objects are “sites of higher level behaviors more appropriate for use as dynamic components” [Kay 1993]

- Intuitively
  - Objects provide *interchangeable services* supporting higher level goals [Aldrich 2013]
What is OOP?

• This is a matter of more debate than expected

• Classically:
  – A combination of data and code
  – Abstraction, Encapsulation, Inheritance, Polymorphism
  – “An object is a value exporting a procedural interface to data or behavior” [Cook 2009]
  – Objects are “sites of higher level behaviors more appropriate for use as dynamic components” [Kay 1993]

• Intuitively
  – Objects provide interchangeabe services supporting higher level goals [Aldrich 2013]
  – You can think of OOP as like writing a library for a task
What is OOP?

- This is a matter of more debate than expected

- Classically:
  - A combination of data and code
  - Abstraction, Encapsulation, Inheritance, Polymorphism
  - “An object is a value exporting a procedural interface to data or behavior” [Cook 2009]
  - Objects are “sites of higher level behaviors more appropriate for use as dynamic components” [Kay 1993]

- Intuitively
  - Objects provide \textit{interchangeable services} supporting higher level goals [Aldrich 2013]
  - You can think of OOP as like writing a library for a task
  - OOP is about decoupling implementation from use
What is OOP?

- This is a matter of more debate than expected
- Classically:
  - A combination of data and code
  - Abstraction, Encapsulation, Inheritance, Polymorphism
  - “An object is a value exporting a procedural interface to data or behavior”
  - Objects are “sites of higher level behaviors more appropriate for use as dynamic components”
- Intuitively
  - Objects provide *interchangeable services* supporting higher level goals
  - You can think of OOP as like writing a library for a task
  - OOP is about decoupling implementation from use
Review of classes (the same exist in Java, .NET, ...)

- Classes describe the services of objects
  - Objects are instances of classes

```cpp
class Student : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };

    Student(Degree degree);

    void studyOneHour();

    void sleep() override;

private:
    int hoursStudied;
    Degree degree;
};
```
Review of classes (the same exist in Java, .NET, ...)

- Classes describe the services of objects
  - Objects are instances of classes
  - Fields define the state an object has

```cpp
class Student : public Person {
    public:
        enum class Degree {
            UNDERGRAD, MASTERS, PHD,
        };

        Student(Degree degree);
        void studyOneHour();
        void sleep() override;

    private:
        int hoursStudied;
        Degree degree;
};
```
• **Classes describe the services of objects**
  – Objects are instances of classes
  – Fields define the state an object has
  – Methods define the behaviors

```cpp
class Student : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };

    Student(Degree degree);
    void studyOneHour();
    void sleep() override;

private:
    int hoursStudied;
    Degree degree;
};
```
Classes describe the services of objects
- Objects are instances of classes
- Fields define the state an object has
- Methods define the behaviors
- Visibility modifiers enable deciding what is published to the outside

```cpp
class Student : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };

    Student(Degree degree);
    void studyOneHour();
    void sleep() override;

private:
    int hoursStudied;
    Degree degree;
};
```
Review of classes (the same exist in Java, .NET, ...)

• Classes describe the services of objects
  – Objects are instances of classes
  – Fields define the state an object has
  – Methods define the behaviors
  – Visibility modifiers enable deciding what is published to the outside
  – Nested constructs enable the use of scoped enums, classes, aliases, ...

```cpp
class Student : public Person {
    public:
        enum class Degree {
            UNDERGRAD, MASTERS, PHD,
        };
        Student(Degree degree);
        void studyOneHour();
        void sleep() override;
    private:
        int hoursStudied;
        Degree degree;
};
```
Review of classes  (the same exist in Java, .NET, ...)

- **Classes describe the services of objects**
  - Objects are instances of classes
  - Fields define the state an object has
  - Methods define the behaviors
  - Visibility modifiers enable deciding what is published to the outside

- Nested constructs enable the use of scoped enums, classes, aliases, ...

- Virtual methods & inheritance enable *derived* classes with the attributes of *base* classes

```cpp
class Student : public Person {
    public:
        enum class Degree {
            UNDERGRAD, MASTERS, PHD,
        };

        Student(Degree degree);
        void studyOneHour();

    private:
        int hoursStudied;
        Degree degree;
};
```
Review of classes (the same exist in Java, .NET, ...)

- **Classes** describe services
  - Objects are instances of classes
  - Fields define the state an object has
  - Methods define the behaviors
  - Visibility modifiers enable deciding what is published to the outside
  - Nested constructs enable the use of scoped enums, classes, aliases, ...
  - Virtual methods & inheritance enable *derived* classes with the attributes of *base* classes

```cpp
class Person {
public:
    Person();
    virtual ~Person() = default;
    virtual void sleep() = 0;
};

class Student : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };
    Student(Degree degree);
    void studyOneHour();
    void sleep() override;
private:
    int hoursStudied;
    Degree degree;
};
```
Review of classes (the same exist in Java, .NET, ...)

- Classes describe objects
  - Objects are instances of classes
  - Fields define the state an object has
  - Methods define the behaviors
  - Visibility modifiers enable deciding what is published to the outside
  - Nested constructs enable the use of scoped enums, classes, aliases, ...
  - Virtual methods & inheritance enable derived classes with the attributes of base classes

```cpp
class Person {
public:
    Person();
    virtual ~Person() = default;
    virtual void sleep() = 0;
};

void processPerson(Person& p);
... Student s{Student::Degree::PHD};
processPerson(s);
```

```cpp
class Student : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };
    Student(Degree degree);
    void studyOneHour();
    void sleep() override;

private:
    int hoursStudied;
    Degree degree;
};
```
General guidelines for classes

- Several guidelines & rules of thumb exist
General guidelines for classes

- Several guidelines & rules of thumb exist
  - Key: Every guideline has a reason.
    Every guideline has exceptions.
    Understand the reason to perform cost-benefit analysis.
General guidelines for classes

- Several guidelines & rules of thumb exist
  - Key: Every guideline has a reason.
    Every guideline has exceptions.
    Understand the reason to perform cost-benefit analysis.

- Most common examples:
  - SOLID
General guidelines for classes

● Several guidelines & rules of thumb exist
  – Key: Every guideline has a reason.
    Every guideline has exceptions.
    Understand the reason to perform cost-benefit analysis.

● Most common examples:
  – SOLID
    • Single Responsibility
    • Open/Closed (more later)
    • Liskov Substitutability
    • Interface Segregation
    • Dependency inversion
General guidelines for classes

- Several guidelines & rules of thumb exist
  - Key: Every guideline has a reason. Every guideline has exceptions. Understand the reason to perform cost-benefit analysis.

- Most common examples:
  - SOLID
    - Single Responsibility
    - Open/Closed (more later)
    - Liskov Substitutability
    - Interface Segregation
    - Dependency inversion
  - DRY (Don’t Repeat Yourself)
General guidelines for classes

• Several guidelines & rules of thumb exist
  – Key: Every guideline has a reason.
    Every guideline has exceptions.
    Understand the reason to perform cost-benefit analysis.

• Most common examples:
  – SOLID
    • Single Responsibility
    • Open/Closed (more later)
    • Liskov Substitutability
    • Interface Segregation
    • Dependency inversion
  – DRY (Don’t Repeat Yourself)
  – These in particular are abused via dogma and misapplication
General guidelines for classes

• Several guidelines & rules of thumb exist
  – Key: Every guideline has a reason.
    Every guideline has exceptions.
    Understand the reason to perform cost-benefit analysis

• Most common examples:
  – SOLID
    • Single Responsibility
    • Open/Closed (more later)
    • Liskov Substitutability
    • Interface Segregation
    • Dependency inversion
  – DRY (Don’t Repeat Yourself)
  – These in particular are abused via dogma and misapplication

All of these relate to Ousterhout’s complexity criteria, but blind application can be worse.
General guidelines for classes (common)

- Be careful about compiler provided methods

```cpp
class Thing {
    // Thing()
    // Thing(const Thing&);
    // Thing(Thing&&);
    // [virtual] ~Thing();
    // Thing& operator==(const Thing&);
    // Thing& operator==(Thing&&);
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability

```cpp
class RGBColor {
public:
  RGBColor(const Intensity r,  
            const Intensity g,  
            const Intensity b);

  Hue convertToHue() const;

private:
  const Intensity red;
  const Intensity green;
  const Intensity blue;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability

```cpp
class RGBColor {
public:
    RGBColor(const Intensity r, const Intensity g, const Intensity b);
    Hue convertToHue() const;

private:
    const Intensity red;
    const Intensity green;
    const Intensity blue;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability

```cpp
class RGBColor {
public:
    RGBColor(const Intensity r,
             const Intensity g,
             const Intensity b);
    Hue convertToHue() const;

private:
    const Intensity red;
    const Intensity green;
    const Intensity blue;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility

```cpp
template<typename T>
class Set {
public:
    Set();
    void insert(const T& toAdd);
    bool contains(const T& toFind) const;

private:
    std::vector<T> elements;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility

```cpp
template<typename T>
class Set {
public:
    Set();
    void insert(const T& toAdd);
    bool contains(const T& toFind) const;

private:
    std::vector<T> elements;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility

```cpp
template<typename T>
class Set {
public:
    Set();
    void insert(const T& toAdd);
    bool contains(const T& toFind) const;

private:
    std::vector<T> elements;
};

struct Point {
    int x;
    int y;
};
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
- Refer to objects by interfaces when applicable
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
- Refer to objects by interfaces when applicable
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
- Refer to objects by interfaces when applicable

```cpp
void printTree(const Tree& tree, const TreeTraversal& t) {
    t.traverse(tree, printNode);
}
```
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
- Refer to objects by interfaces when applicable
- Don’t give away your internals
General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability
- Minimize visibility
- Refer to objects by interfaces when applicable
- Don’t give away your internals

```cpp
class IntBuffer {
public:
    ... std::vector& getContents(); ...
private:
    std::vector<int> integers;
};
```
General guidelines for classes

- Prefer dependency injection to hardwiring resources [Block 2001, 2018]
  - Objects that allocate their own state are hard to:
    prove correct, extend, configure, test, ...
General guidelines for classes

- Prefer dependency injection to hardwiring resources [Block 2001,2018]
  - Objects that allocate their own state are hard to:
    prove correct, extend, configure, test, ...

```cpp
class CrosswordGenerator {
    CrosswordGenerator()
        : clues{std::make_unique<Clues>}
        { }

private:
    std::unique_ptr<Clues> clues;
};
```
General guidelines for classes

- Prefer dependency injection to hardwiring resources [Block 2001, 2018]
  - Objects that allocate their own state are hard to: prove correct, extend, configure, test, ...

```cpp
class CrosswordGenerator {
    CrosswordGenerator() :
        clues{std::make_unique<Clues>}
    { }

private:
    std::unique_ptr<Clues> clues;
};

class CrosswordGenerator {
    CrosswordGenerator(... clues)
        : clues{std::move(clues)}
    { }

private:
    std::unique_ptr<Clues> clues;
};
```
General guidelines for classes

- Prefer dependency injection to hardwiring resources [Block 2001,2018]
  - Objects that allocate their own state are hard to:
    prove correct, extend, configure, test, ...

```cpp
class CrosswordGenerator {
  CrosswordGenerator() :
    clues{std::make_unique<Clues>}
    { }
  private:
    std::unique_ptr<Clues> clues;
};

auto englishClues = ...
CrosswordGenerator cg{englishClues};

auto frenchClues = ...
CrosswordGenerator cg{frenchClues};
```
General guidelines for classes

- **Prefer dependency injection to hardwiring resources** [Block 2001,2018]
  - Objects that allocate their own state are hard to:
    prove correct, extend, configure, test, ...

```cpp
class CrosswordGenerator {
    CrosswordGenerator()
        : clues{std::make_unique<Clues>}
    {
    }

private:
    std::unique_ptr<Clues> clues;
};

auto englishClues = ...
CrosswordGenerator cg{englishClues};
```

Separating the *creation* of objects from the *wiring* of objects creates a more flexible system

```cpp
auto frenchClues = ...
CrosswordGenerator cg{frenchClues};
```
General guidelines for classes

- Some are specific to “native code”:
  - Use the PIMPL idiom judiciously [Sutter & Alexandrescu 2005]
    - Prevents unnecessary recompilation
    - Allows the layout to change without breaking ABI in long lived projects
Some are specific to “native code”:
Use the PIMPL idiom judiciously [Sutter & Alexandrescu 2005]
- Prevents unnecessary recompilation
- Allows the layout to change without breaking ABI in long lived projects

```cpp
class Thing {
public:
  Thing();

  void doStuff() const;

private:
  class ThingImpl;
  std::unique_ptr<ThingImpl> impl;
};
```
General guidelines for classes

- Some are specific to “native code”:
  Use the PIMPL idiom judiciously [Sutter & Alexandrescu 2005]
  - Prevents unnecessary recompilation
  - Allows the layout to change without breaking ABI in long lived projects

```cpp
class Thing {
public:
  Thing();
  void doStuff() const;

private:
  class ThingImpl;
  std::unique_ptr<ThingImpl> impl;
};

Thing.h

Thing.cpp

Thing::Thing() :
    impl{std::make_unique<ThingImpl>()} {
}

void
Thing::doStuff() const {
    impl->doStuff();
}
```
Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface

```
List
+ begin()
+ end()
+ push_back()
+ clear()
```
Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface
  - Derived classes provide implementations

```
List
  + begin()
  + end()
  + push_back()
  + clear()

ArrayList
  + begin()
  + end()
  + push_back()
  + clear()

LinkedList
  + begin()
  + end()
  + push_back()
  + clear()
```
Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface
  - Derived classes provide implementations
  - Implementations are interchangeable even at runtime (like remote services)
Thinking in terms of services

- Modern thinking notes that OOP defines services:
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface
  - Derived classes provide implementations
  - Implementations are interchangeable even at runtime (like remote services)

```cpp
void transferStudents(List<Student>& from, List<Student>& to) {
    ranges::copy(from, std::back_inserter(to));
    from.clear();
}
```
Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface
  - Derived classes provide implementations
  - Implementations are interchangeable even at runtime (like remote services)

- This also enables heterogeneous aggregates

```cpp
void letThePeopleSleep(List<Person*>& people) {
    for (Person* person : people) {
        person->sleep();
    }
}
```
So let’s try it out...

Note: We will go from absurd to practical
So let’s try it out...

- Suppose we want to model a person who owns a car...
So let’s try it out...

- Suppose we want to model a person who owns a car...
So let’s try it out...

- Suppose we want to model a person who owns a car...

```cpp
class CarOwner : public Person, Car
{
};
```
So let’s try it out...

- Suppose we want to model a person who owns a car...

```
class CarOwner : public Person, Car
{
};
```

Is this good or bad? Why?
So let’s try it out...

- Suppose we want to model a person who owns a car...

```
class CarOwner : public Person, Car {
};
```

Is this good or bad? Why?

How could you make it better?
So let’s try it out...

- Suppose we want to model a person who owns a car...

```
Person       Car
  is-a        is-a
CarOwner

Person       Car
  is-a       has-a
CarOwner     Car
```
So let’s try it out...

- Suppose we want to model a person who owns a car...

Even simpler?
So let’s try it out...

- Suppose we want to model a person who owns a car...
So let’s try it out...

- Suppose we want to model a person who owns a car...

That a car is amongst a person’s possessions does not make them a special person.
Suppose we want to model a person who owns a car...

That a car is amongst a person’s possessions does not make them a special person.

This absurd example captures common, subtle mistakes.
So why is inheritance hard?

- Do the LSP and has-a relationships *unambiguously* tell us how to apply inheritance?
So why is inheritance hard?

- Do the LSP and has-a relationships *unambiguously* tell us how to apply inheritance?

*Frogs can be male or female*
So why is inheritance hard?

- Do the LSP and has-a relationships *unambiguously* tell us how to apply inheritance?

- Frogs can be male or female

- Diagram of frog inheritance:
  - Frog
  - MaleFrog
  - FemaleFrog
So why is inheritance hard?

- Do the LSP and has-a relationships *unambiguously* tell us how to apply inheritance?

Frogs can be male or female

- Frog
  - MaleFrog
  - FemaleFrog

Frog
- sex:{male,female}
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?
- Every *is-a* relationship could instead be *has-a*!
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Professor
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every *is-a* relationship could instead be *has-a*!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

`Professor` \( \text{has-a} \) `Researcher`
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?
- Every *is-a* relationship could instead be *has-a*!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

![Diagram showing relationships between Professor, Researcher, and Teacher]
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every \textit{is-a} relationship could instead be \textit{has-a}!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every *is-a* relationship could instead be *has-a*!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Note, these are now *roles*, not *people*.
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Note, these are now roles, not people.

- Whenever is-a applies, you must still make a decision
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Coarse inheritance often precludes it
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

Frogs and other animals can spontaneously change sex!
Choosing is-a or has-a

- **Guide 1: Might the behavior need to change?**
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

Frogs and other animals can spontaneously change sex!

Knowing in advance is hard. Composition is flexible & adapts to requirements.
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

- **Guide 2:** Might the type be used *polymorphically*?
  - Composition does not intrinsically aid it
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

- **Guide 2:** Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
Choosing is-a or has-a

- **Guide 1: Might the behavior need to change?**
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

- **Guide 2: Might the type be used polymorphically?**
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one.
Choosing is-a or has-a

- **Guide 1**: Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

- **Guide 2**: Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one.

```cpp
std::vector<People*> folks;
```

0) Student
1) Student
2) Lecturer
3) Professor
4) Student
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Use coarse grained composition if the relationship is dynamic

- **Guide 2:** Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance enables it
  - Consider inheritance when a reference to a general type may point to a more specific one.

```
std::vector<People*> folks;
```

We will revisit this in the context of *algebraic data types.*
So let’s try it out...

- I need
  - Many different types of animals.

This should sound familiar...
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `{move()} and `{speak()`. 
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

What does my design look like based on the rules?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

```
Animal
  /   \
Parrot Cat Professor Corgi
  / \
Maine Coon Bengal
```

Is this good?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Does Cat serve a purpose?

Is this good?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Is this good?
Does it achieve reuse?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Is this good?
Does it achieve reuse?
What if I want a new Animal at run time?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to \texttt{move()} and \texttt{speak()}.
  - An \texttt{Animal} should be able to refer to any of them.

Can we do better?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Can we do better?

If someone on my team did this multiple times, I would fire them.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal &` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change
I need

- Many different types of animals.
- Each should be able to `move()` and `speak()`.
- An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

![Diagram showing relationships between Animal and Movement]
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?  Recall: identify & isolate change

Animal has-a Movement

Movement selects from the ways any Animal can move.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?  
Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?  
Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to move() and speak().
  - An Animal should be able to refer to any of them.

Can we do better?  Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to move() and speak().
  - An Animal should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?
Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to \texttt{move()} and \texttt{Speak()}.
  - An \texttt{Animal} should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

- Animal
  - Movement
    - Crawl
    - Fly
    - Saunter
  - Vocalization
    - Tweet
    - Meow
    - Ramble
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change

```
Animal
  has-a Movement
    Crawl    Fly    Saunter
  has-a Vocalization
    Tweet    Meow    Ramble    Bark
```
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```java
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```

```
class Movement {
    void move() {
        // Movement logic
    }
}
```

```
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```
So let’s try it out...

- So let’s try it out...(!)
Shallow, fine grained inheritance

- Avoids reimplementaion of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary

Previously static requirements will often become dynamic.
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies

- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary

- Directly identifies & addresses risks of change in class design
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary
- Directly identifies & addresses risks of change in class design
- We will see shortly how this interacts with other forms of polymorphism
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.

For some reason, textbooks & teachers often get these wrong
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.
Guidelines for inheritance

- Favor composition over inheritance
- Do not inherit to reuse. Inherit to be reused.
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If $\phi$ is true for the base, then $\phi$ is true the derived

- Derived is *substitutable* for Base
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general

B <: D

Base
A foo(B b)

Derived
C foo(D d)

Arguments are *contravariant*
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If \( \varphi \) is true for the base, then \( \varphi \) is true for the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

\[
\text{Base} \quad A \quad \text{foo}(B \ b) \\
\text{Derived} \quad C \quad \text{foo}(D \ d) \\
\]

Return types are *covariant*
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
  - *Semantic* substitutability is robust to drift
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
  - *Semantic* substitutability is robust to drift
Guidelines for inheritance

- Inherit interfaces. Push implementation into the leaves.

class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
Guidelines for inheritance

- Inherit interfaces. Push implementation into the leaves.
  - Hierarchies delocalize code, yielding a yo-yo effect
  - Ambiguous overrides break encapsulation

```cpp
class Parent {
  virtual void foo() { bar(); }
  virtual void bar() {}
};
```

[Bloch, “Effective Java”]
Guidelines for inheritance

- Inherit interfaces. Push implementation into the leaves.
  - Hierarchies delocalize code, yielding a yo-yo effect
  - Ambiguous overrides break encapsulation

```cpp
class Parent {
    virtual void foo() { bar(); }
    virtual void bar() {}
};

class Child : public Parent {
public:
    virtual void bar() { foo(); }
};

[Bloch, “Effective Java”]
```
Guidelines for inheritance

- Inherit interfaces. Push implementation into the leaves.
  - Hierarchies delocalize code, yielding a yo-yo effect
  - Ambiguous overrides break encapsulation

```cpp
class Parent {
  virtual void foo() { bar(); }
  virtual void bar() {}  
};

class Child : public Parent {
  public:
    virtual void bar() { foo(); }
};
```

[Bloch, “Effective Java”]
Guidelines for inheritance

- Inherit interfaces. Push implementation into the leaves.
  - Hierarchies delocalize code, yielding a yo-yo effect
  - Ambiguous overrides break encapsulation

```cpp
class Parent {
    public:
        virtual void foo() { bar(); }
        virtual void bar() {};
    virtual void barImpl() = 0;
};

class Child : public Parent {
    public:
        virtual void bar() { foo(); }
};
```

Non Virtual Interfaces (NVI) help clarify & are common in C++.

Other patterns help even more...

[Bloch, “Effective Java”]
Guidelines for inheritance


```cpp
class Student final : public Person {
public:
    enum class Degree {
        UNDERGRAD, MASTERS, PHD,
    };

    Student(Degree degree);

    void studyOneHour();

    void sleep() override;

private:
    int hoursStudied;
    Degree degree;
};
```
Summary

• Object oriented programming is a useful tool in your toolbox
Summary

- Object oriented programming is a useful tool in your toolbox
- It can be challenging to use well and should be deliberate
Summary

- Object oriented programming is a useful tool in your toolbox
- It can be challenging to use well and should be deliberate
- Inheritance, specifically, is powerful but often abused
Summary

- Object oriented programming is a useful tool in your toolbox.
- It can be challenging to use well and should be deliberate.
- Inheritance, specifically, is powerful but often abused.
- **Object orientation does not solve problems in modeling that requires more effort, as we will see.**