CMPT 373
Software Development Methods

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
- Our goal
  - 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 *interchangeable 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 *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" [Cook 2009]
  - Object... for use as dynam...

Consider out maze prototyping example.
Have we already seen *one* way this can be useful?

- Intuitively
  - Objects provide *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

Review of classes (the same exist in Java, .NET, ...)

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

```
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

```
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

```
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

```
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, …

```
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

```
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 ... s
  - Obje...
  - Field...
  - Meth...
  - Visib... t 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

```
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
  - Obje
  - Field
  - Meth
  - Visib
    is published to the outside
  - Nes
    sco

```
class Person {
public:
  Person();
  virtual ~Person() = default;

  virtual void sleep() = 0;
};
```

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

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

```
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
    - **S**ingle Responsibility
    - **O**pen/Closed (more later)
    - **L**iskov Substitutability
    - **I**nterface Segregation
    - **D**ependency 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
    - **S**ingle Responsibility
    - **O**pen/Closed (more later)
    - **L**iskov Substitutability
    - **I**nterface Segregation
    - **D**ependency inversion
  - DRY (**D**on’t **R**epeat **Y**ourself)

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
    - **S**ingle Responsibility
    - **O**pen/Closed (more later)
    - **L**iskov Substitutability
    - **I**nterface Segregation
    - **D**ependency 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 ex
  - 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

```
class Thing {
   // Thing()
}
```

```
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

```
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

```
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

```
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;
};
```

```
const RGBColor color = ...;
```

vs.

General guidelines for classes (common)

- Be careful about compiler provided methods
- Minimize mutability

```
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

```
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

```
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

```
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

```
struct Point {
  int x;
  int y;
};
```

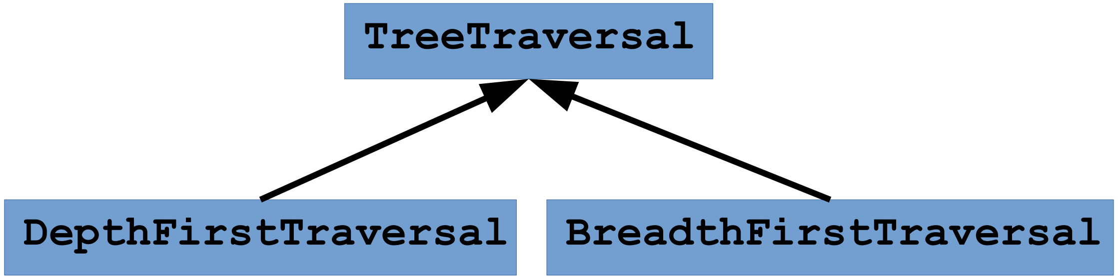
```
template<typename T>
class Set {
public:
  Set();

  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
- Refer to objects by interfaces when applicable

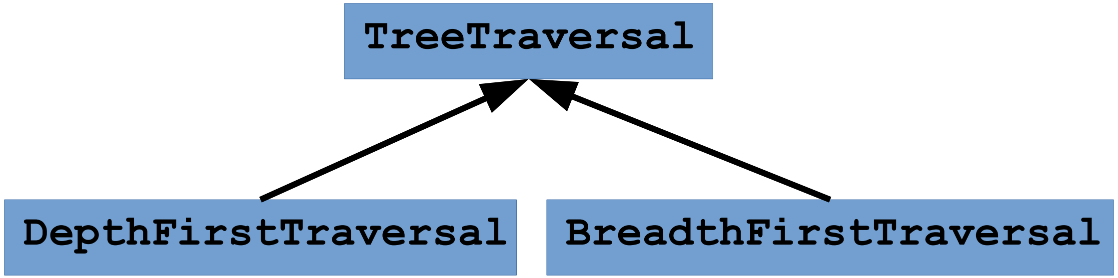
General guidelines for classes (common)

- 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




```
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

```
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, …

```
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, ...

```
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, ...

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

private:
  std::unique_ptr<
};
```

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

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

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

```
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, ...

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

private:
  std::unique_ptr<
};
```

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

```
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

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

Thing.h

```
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

Thing.h

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

  void doStuff() const;

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

Thing.cpp

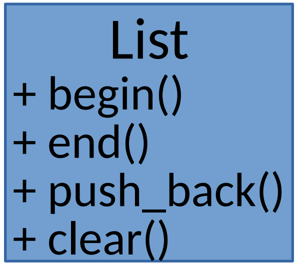
```
Thing::Thing()
  : impl{std::make_unique<ThingImpl>()}
    { }

void
Thing::doStuff() const {
  impl->doStuff();
}
```

Thinking in terms of services

- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this

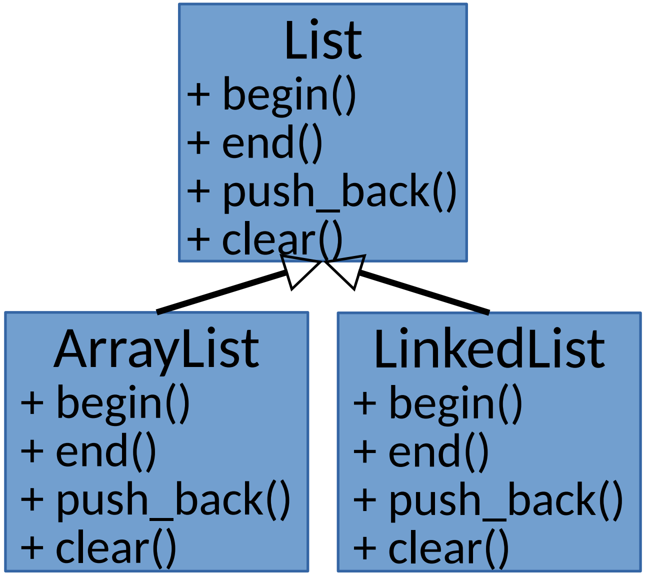
Thinking in terms of services

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



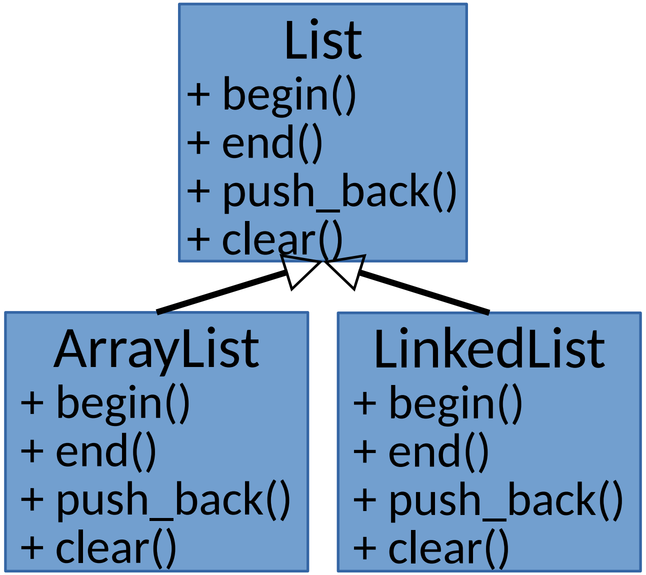
Thinking in terms of services

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



Thinking in terms of services

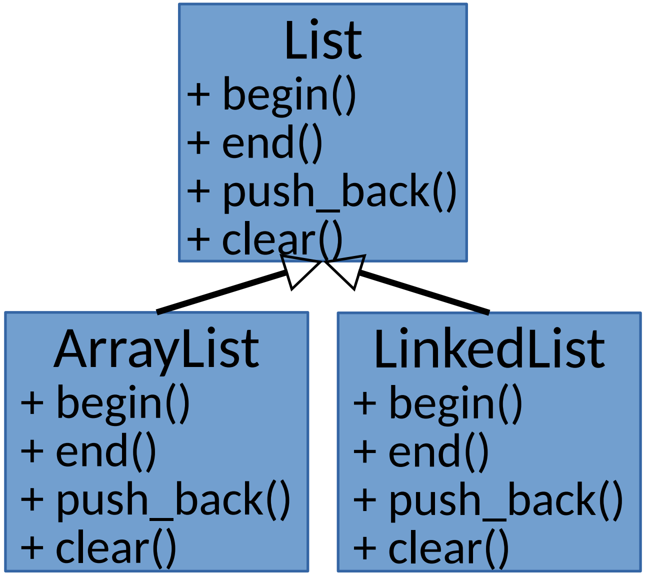
- Modern thinking notes that OOP defines services
  - Inheritance & runtime polymorphism drive this
  - Base classes define an interface
  - Derived classes provide implementations
  - Implementations are interchangeable even at runtime (like remote services)



Thinking in terms of services

- Mod
  - I
  - E
  - D
  - Implementations are interchangeable even at runtime (like remote services)

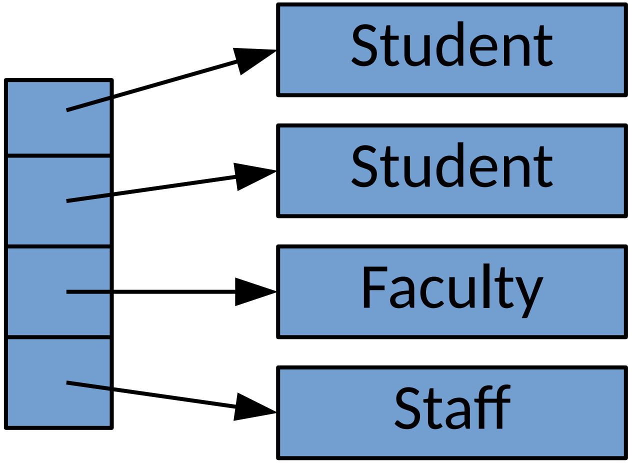
```
void
transferStudents(List<Student>& from, List<Student>& to) {
    ranges::copy(from, std::back_inserter(to));
    from.clear();
}
```



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

```
void
letThePeopleSleep(List<Person*>& people) {
  for (Person* person : people) {
    person->sleep();
  }
}
```



So let's try it out…

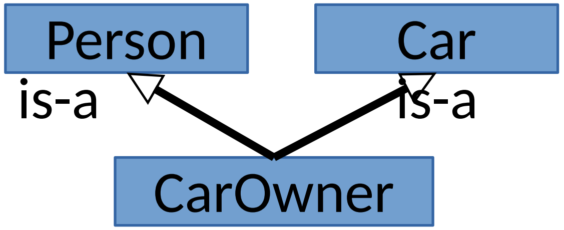
Note: We will go from absurd to practical

So let’s try it out…

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

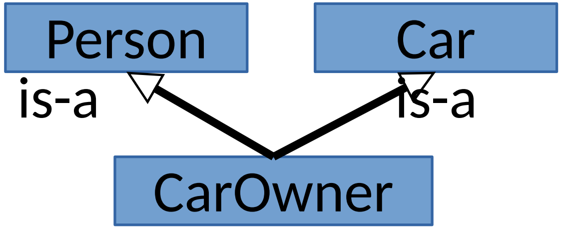
So let's try it out…

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



So let's try it out…

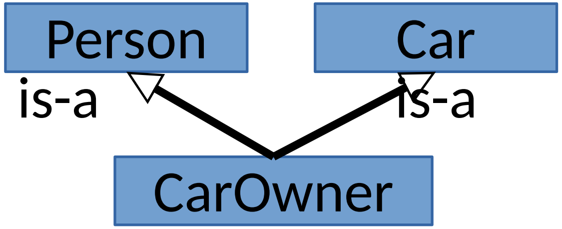
- Suppose we want to model a person who owns a car…




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

So let's try it out…

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

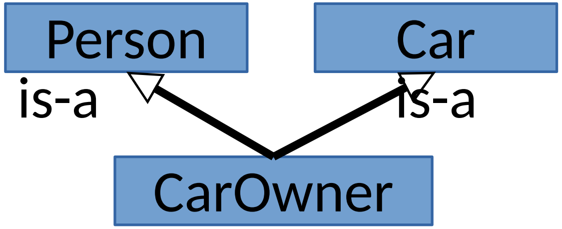



```
class CarOwner : public Person, Car
   { };
```

Is this good or bad?
Why?

So let’s try it out…

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



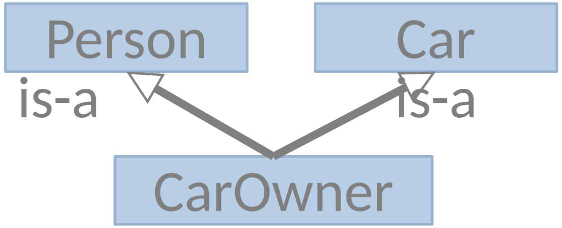

```
class CarOwner : public Person, Car
   { };
```

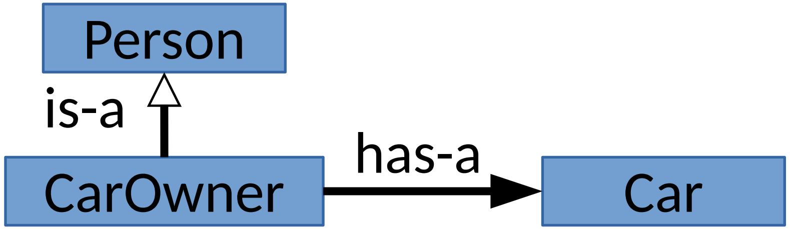
Is this good or bad?
Why?

How could you make it better?

So let's try it out…

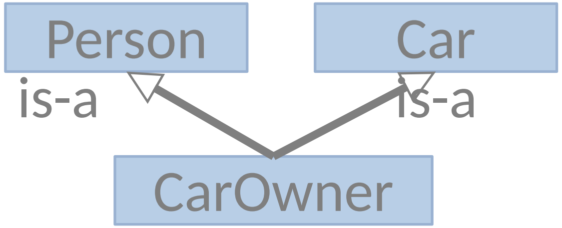
- Suppose we want to model a person who owns a car…

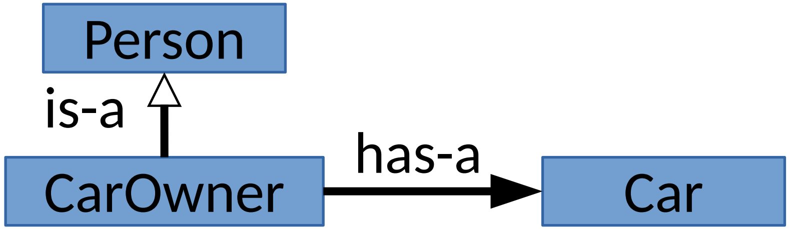




So let's try it out...

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

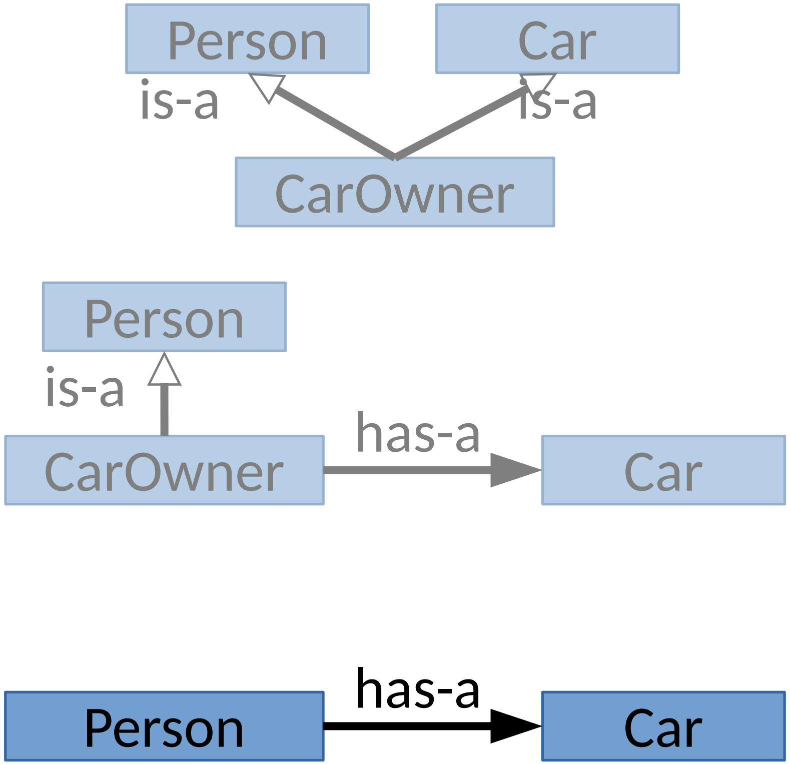




Even simpler?

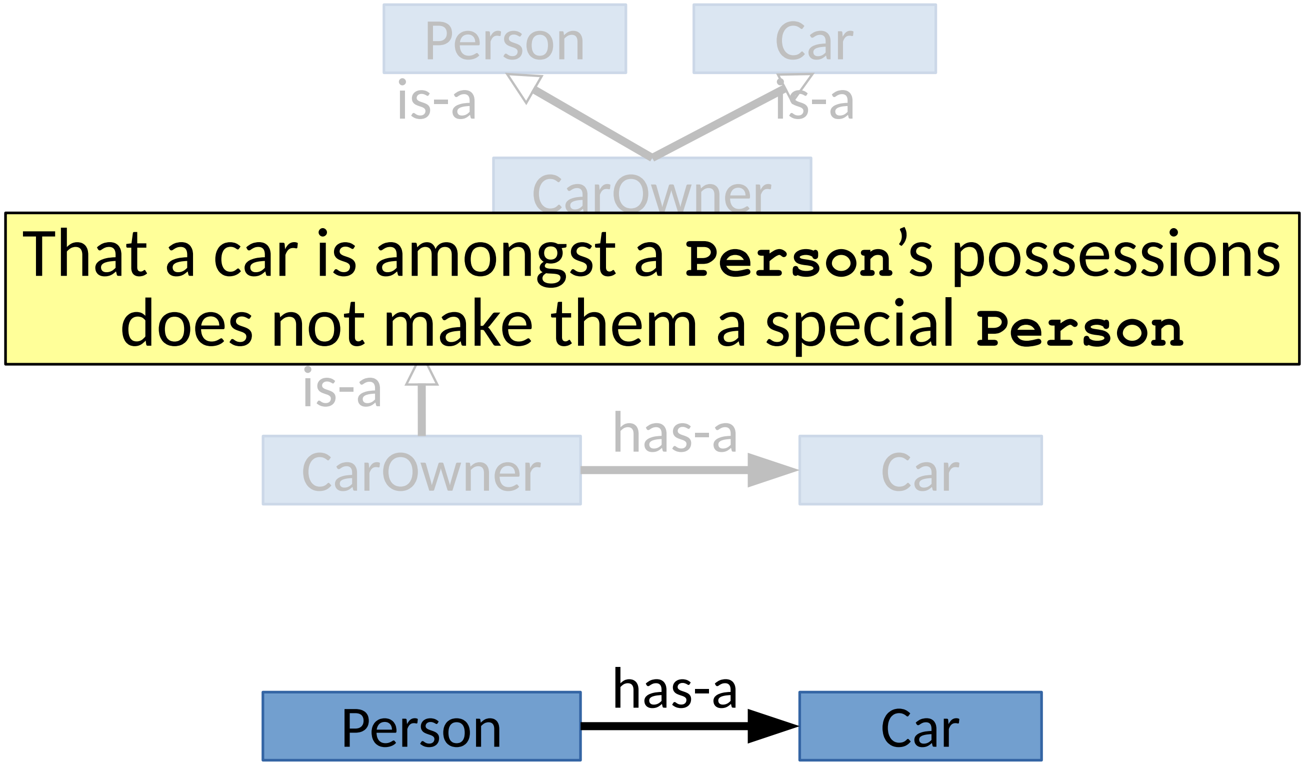
So let’s try it out…

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



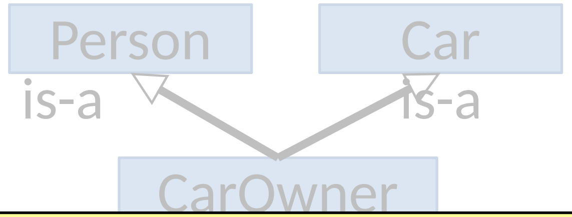
So let's try it out…

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



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`**

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?

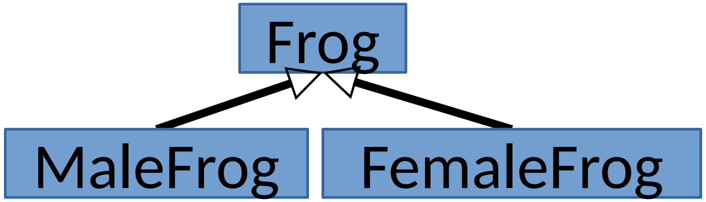
- Do the *LSP* and *has-a* relationships *unambiguously* tell us how to apply inheritance?

`Frog`s can be male or female

So why is inheritance hard?

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

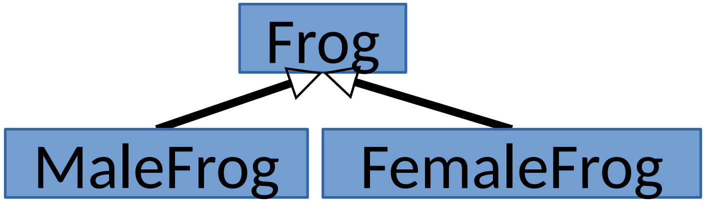
`Frog`s can be male or female

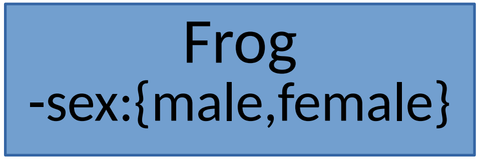


So why is inheritance hard?

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

`Frog`s can be male or 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?

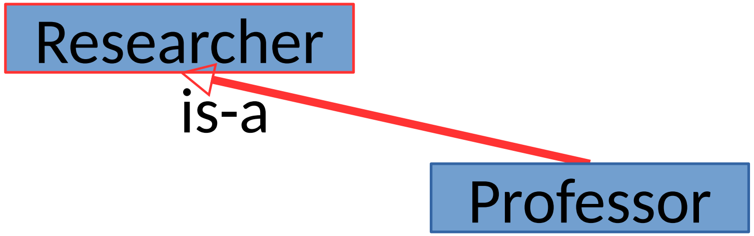
- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?

- 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?

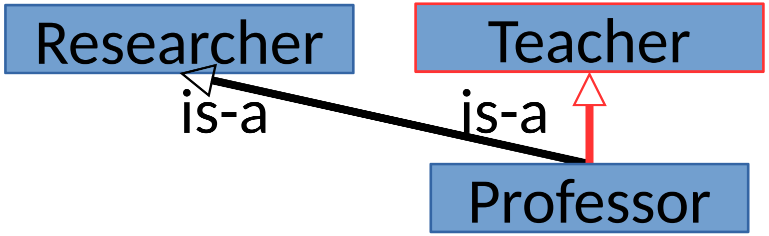
- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



So why is inheritance hard?

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

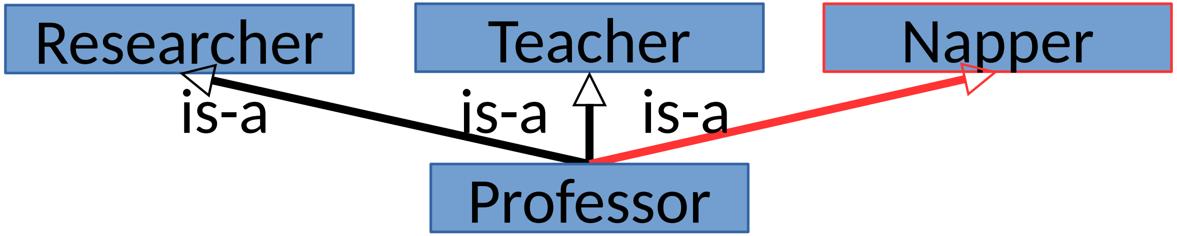
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



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?

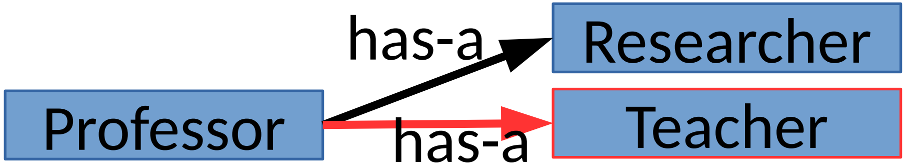
- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?

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

So why is inheritance hard?

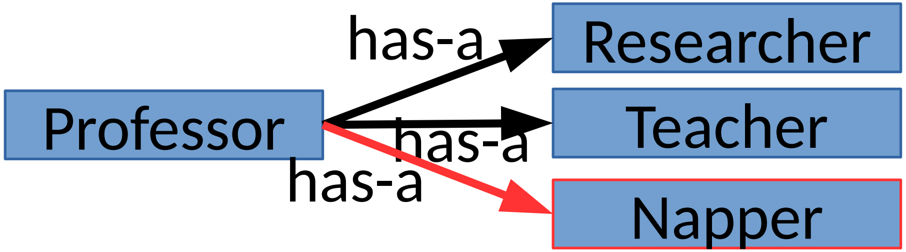
- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?

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



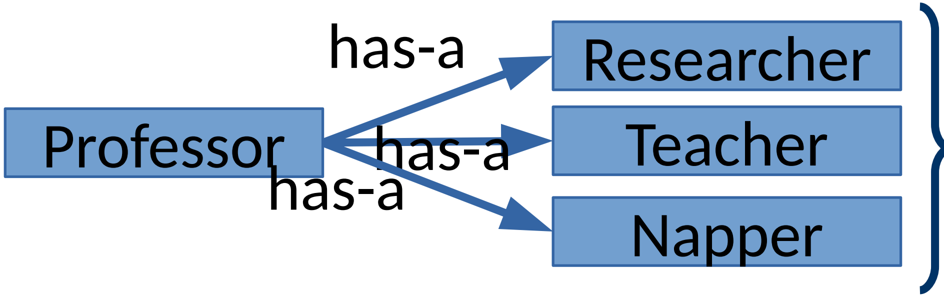
So why is inheritance hard?

- 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?

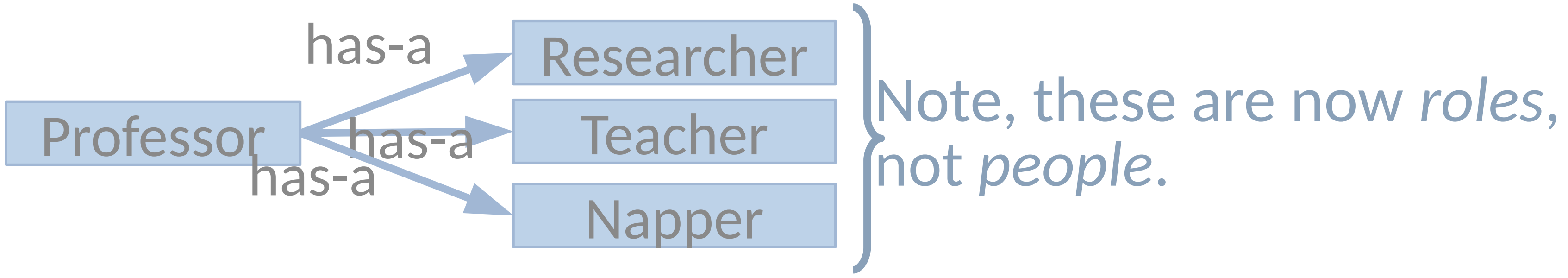
- Do the *LSP* and *has-a* relationships unambiguously tell us how to apply inheritance?
- Every ***is-a*** relationship could instead be ***has-a***!
  - These often capture finer grained relationships
  - Break individual responsibilities into components



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



- **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

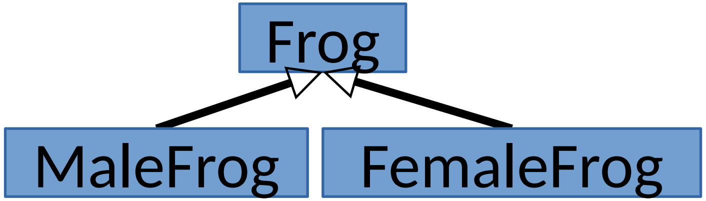
- Guide 1: Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often *simplifies* it

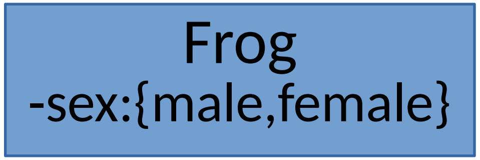
Choosing is-a or has-a

- 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

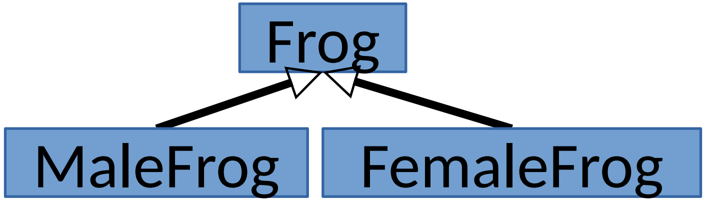
- Guide 1: Might the behavior need to change?
  - Coarse inheritance often precludes it
  - Composition often simplifies it
  - Use coarse grained composition if the relationship is dynamic

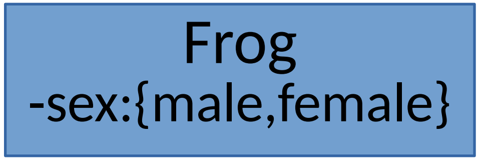




Choosing is-a or has-a

- 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

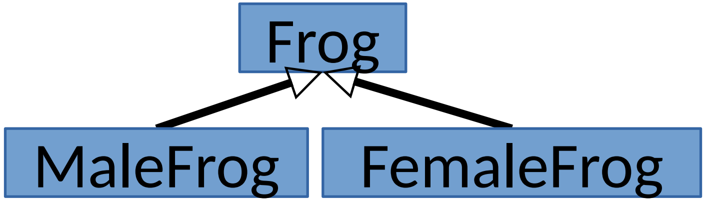


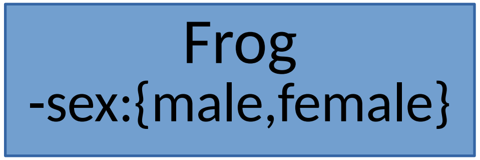


`Frog`s and other animals
can spontaneously change sex!

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





`Frog`s 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
  - C
  - Use coarse grained composition if the relationship is

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

0) Student
1) Student
2) Lecturer
3) Professor
4) Student

- 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
  - C
  - Use coarse grained composition if the relationship is

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

0) Student
1) Student
2) Lecturer
3) Professor
4) Student

- 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.

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.

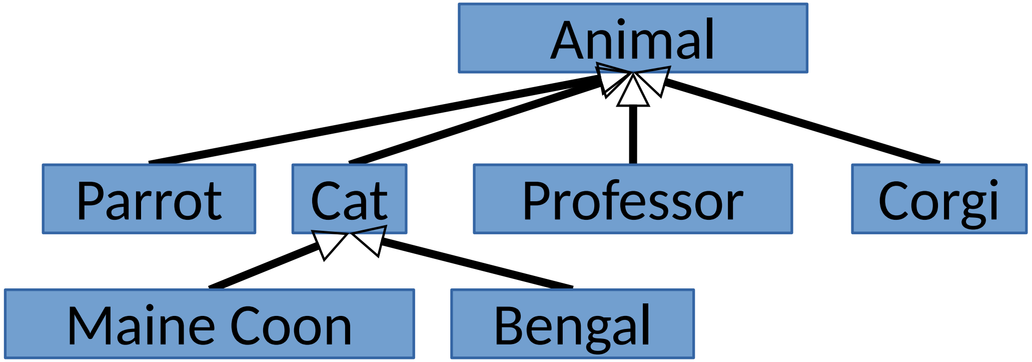
So let's try it out…

- 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?

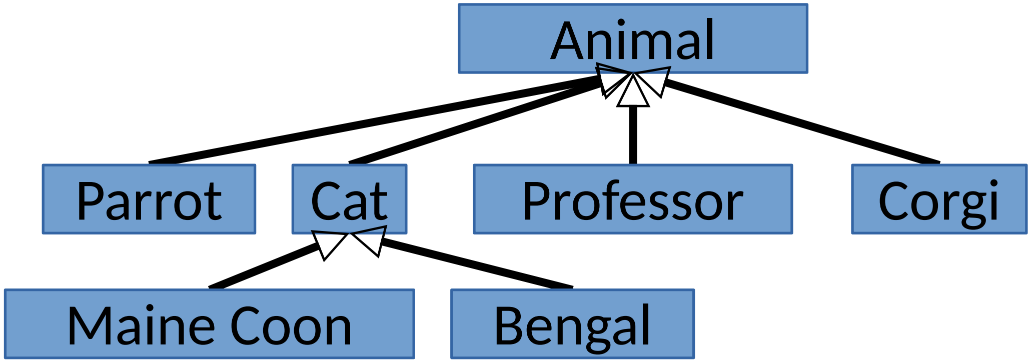
So let's try it out...

- 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.

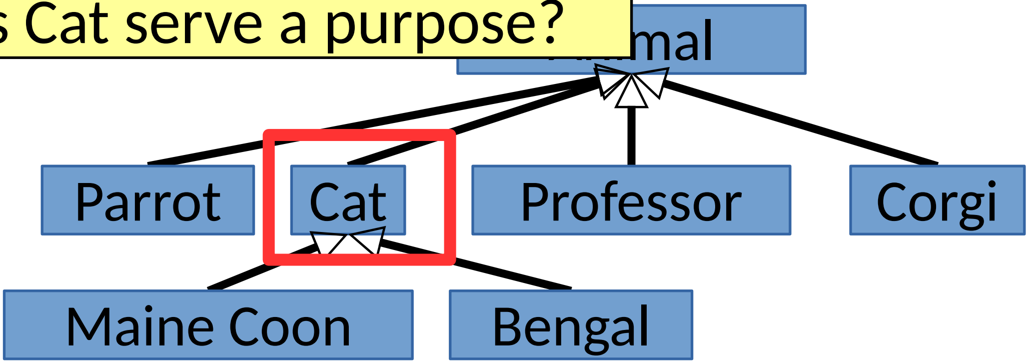


Is this good?

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.

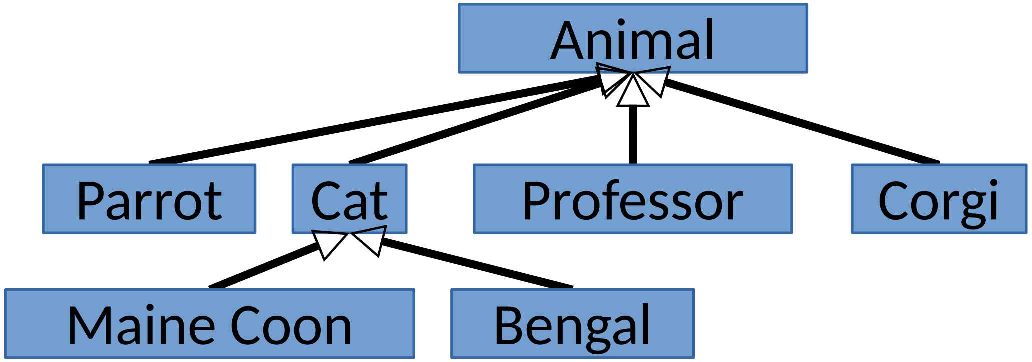
Does Cat serve a purpose?



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.

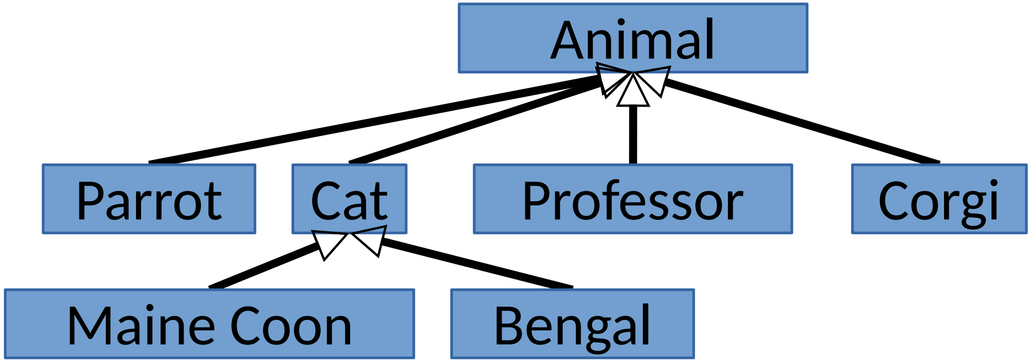


Is this good?

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 `move()` and `speak()`.
  - An `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?

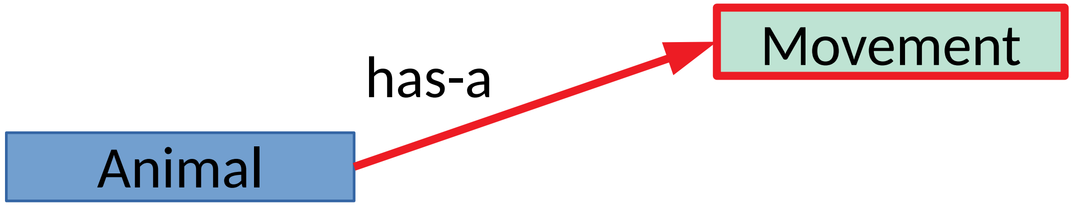
Recall: identify & isolate change

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

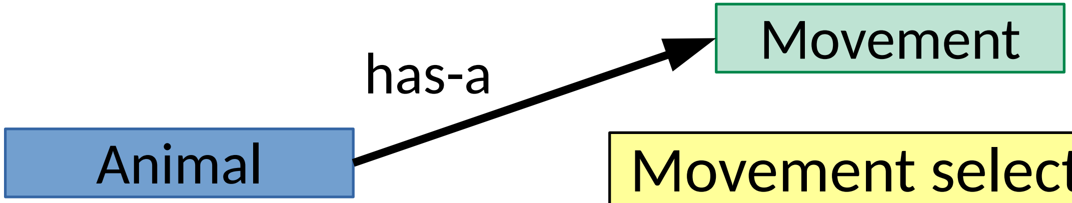


So let's try it out...

- 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



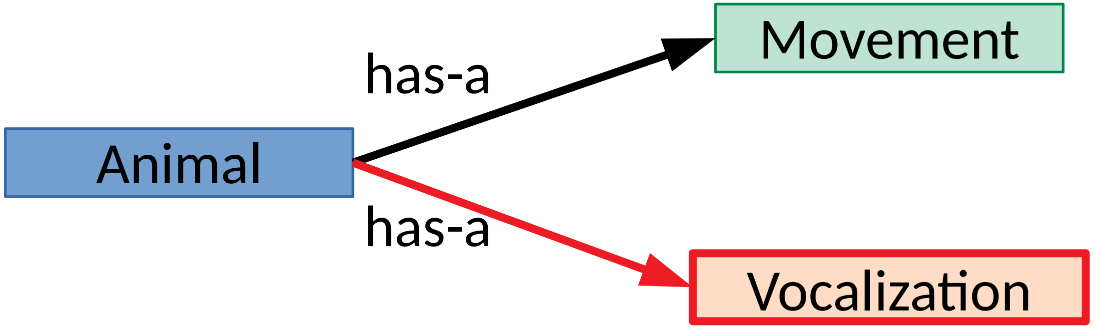
Movement selects from the ways any Animal can move.

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

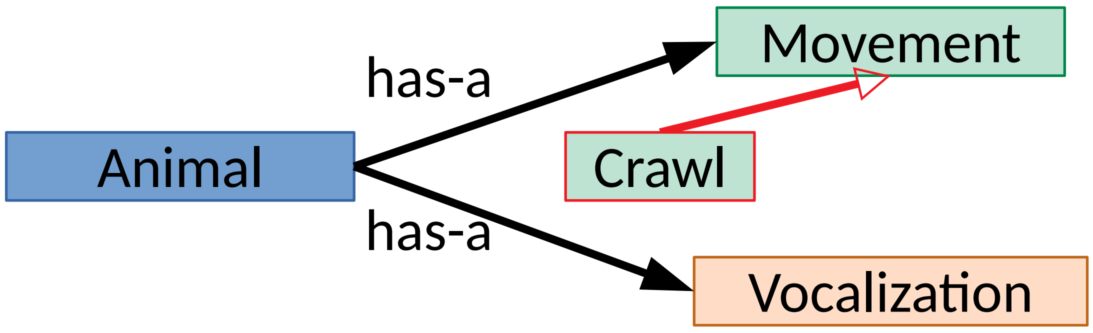


So let's try it out...

- 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

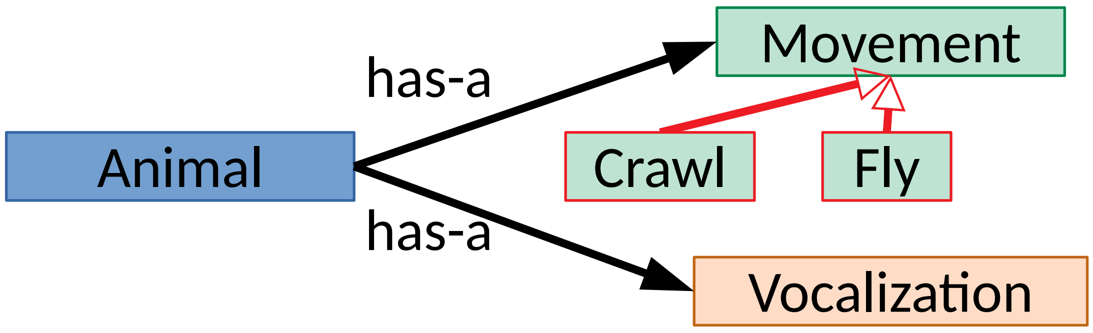


So let's try it out...

- 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

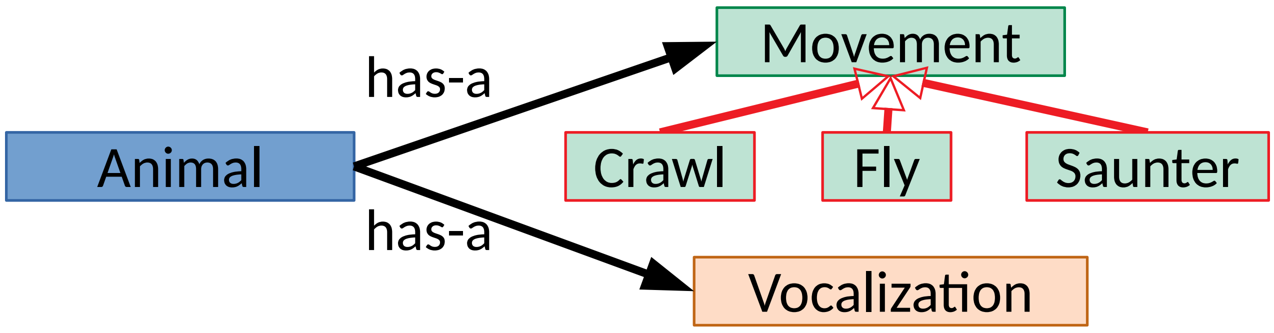


So let's try it out…

- 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

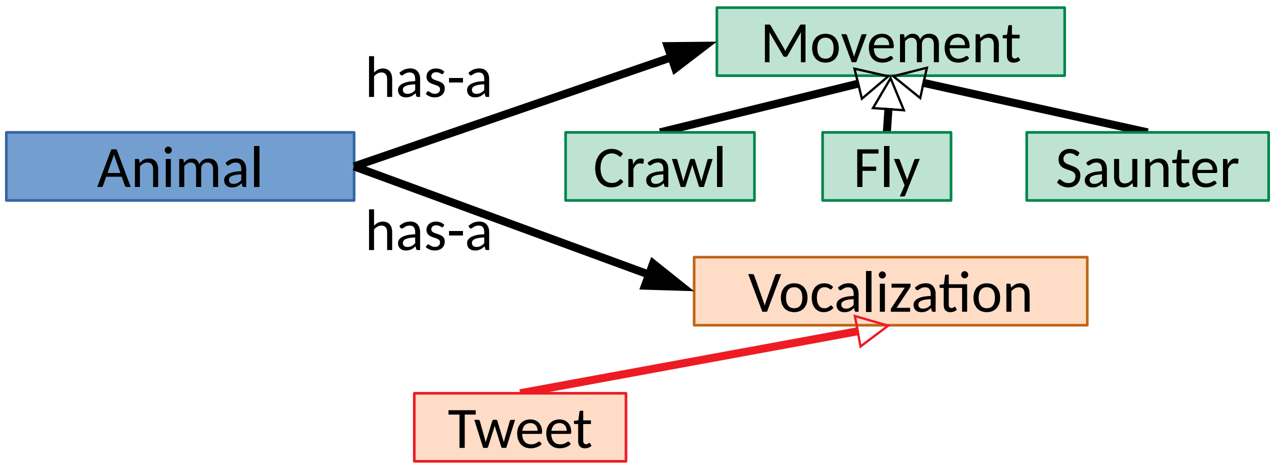


So let’s try it out…

- 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

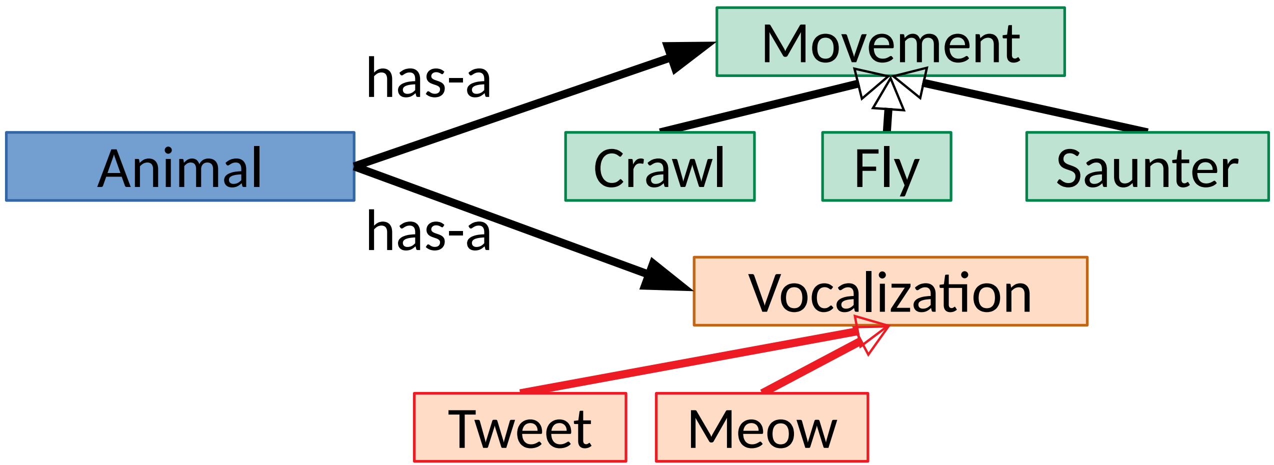


So let’s try it out…

- 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

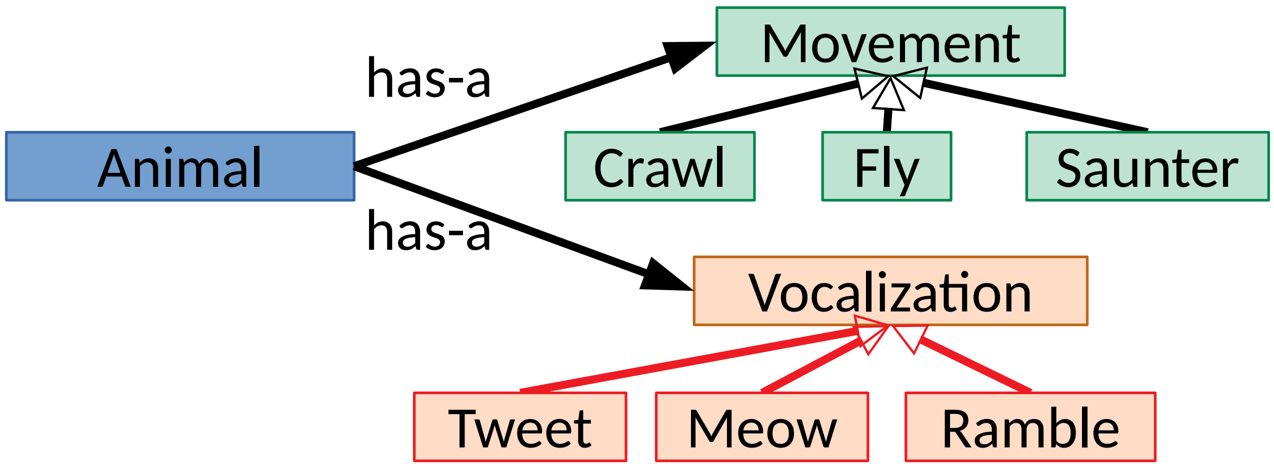


So let's try it out...

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

Can we do better?

Recall: identify & isolate change

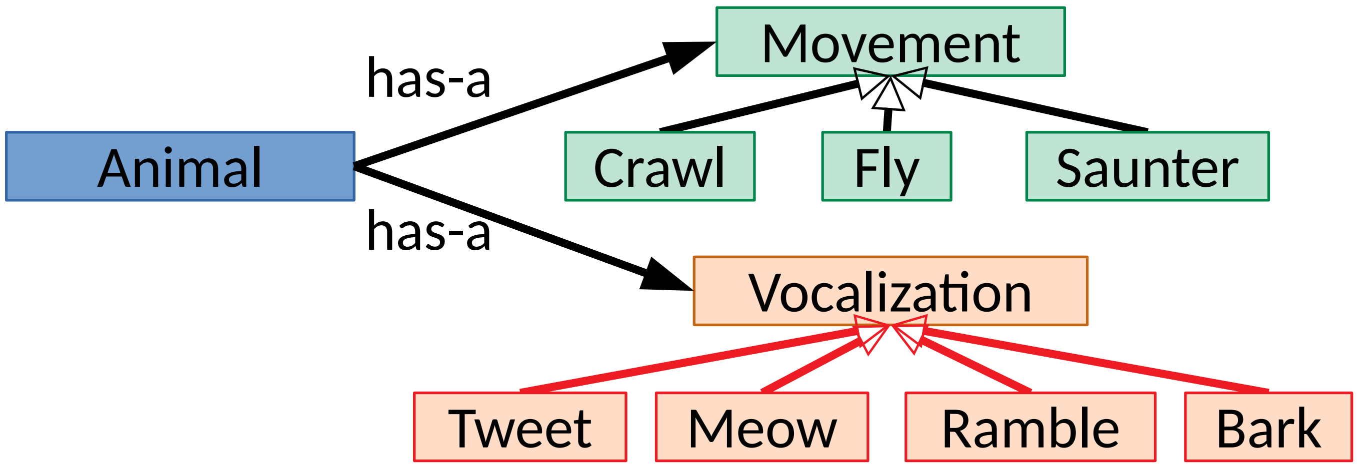


So let's try it out…

- 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



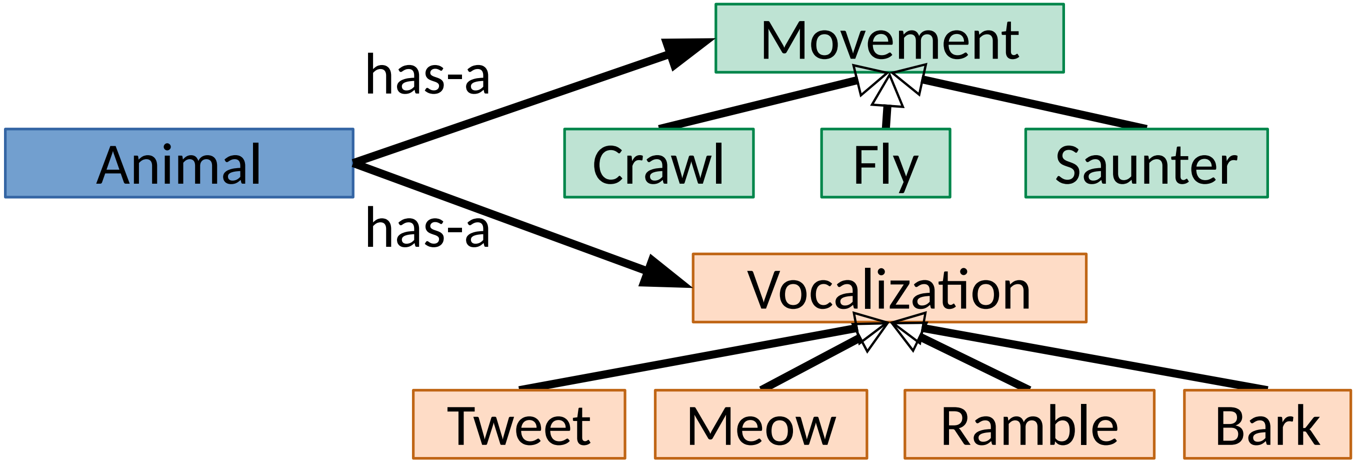
So let's try it out…

- 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

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



So let's try it out…

- So let's try it out…**(!)**

Shallow, fine grained inheritance

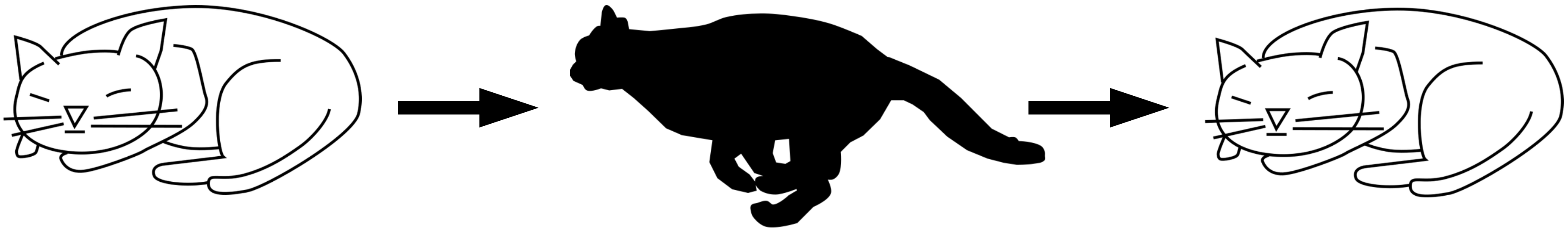
- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

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

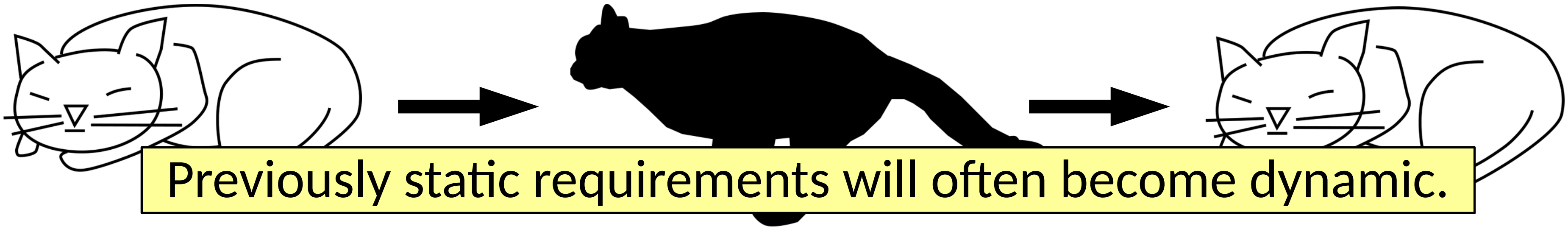
Shallow, fine grained inheritance

- 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.

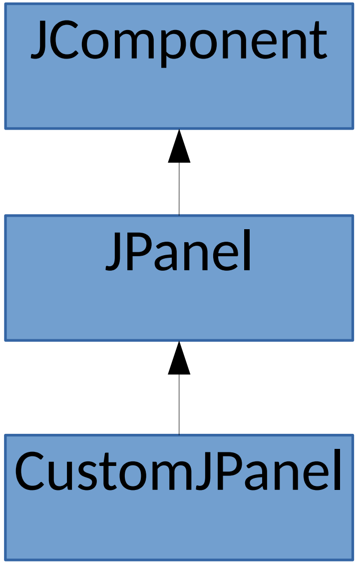
For some reason, textbooks & teachers often get these wrong

Guidelines for inheritance

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

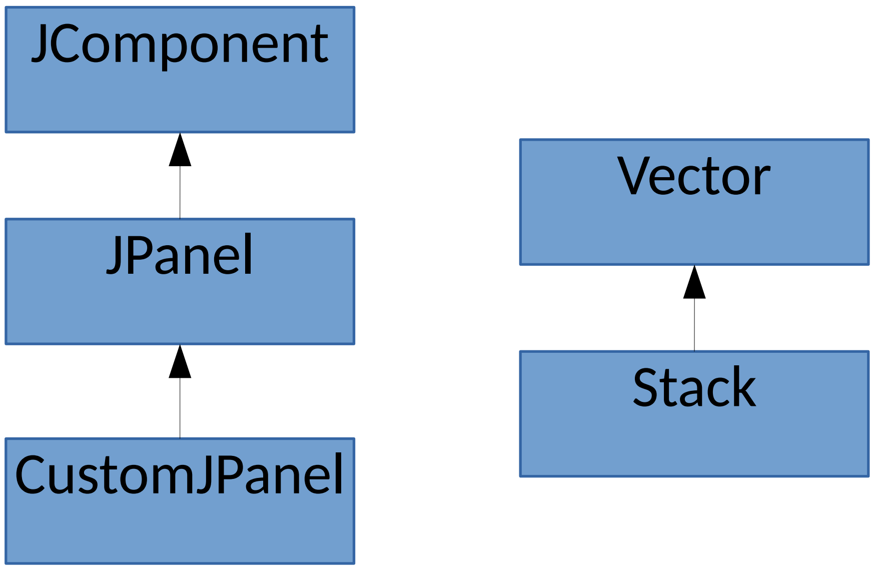
Guidelines for inheritance

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



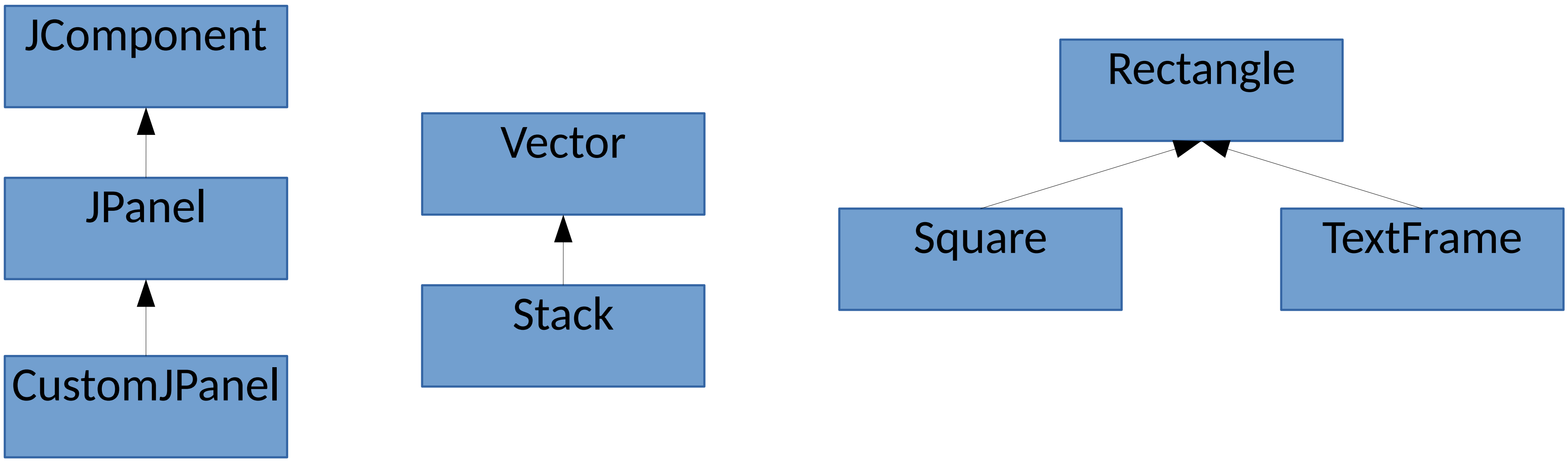
Guidelines for inheritance

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



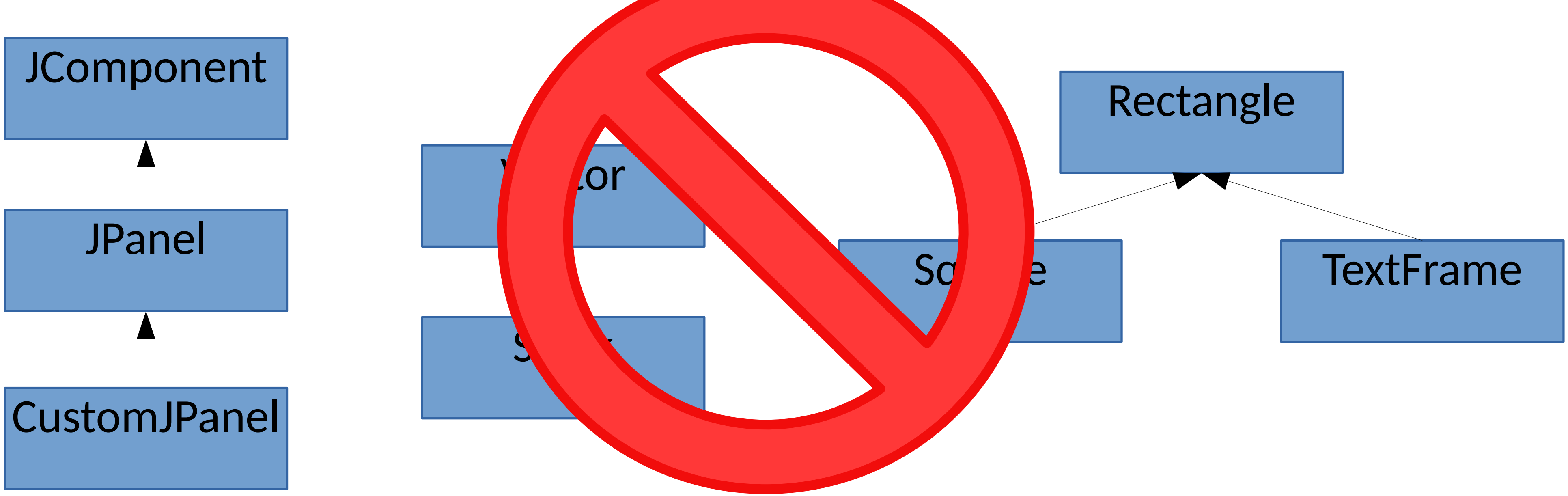
Guidelines for inheritance

- 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

Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability

Guidelines for inheritance

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

Derived is ***substitutable*** for Base

Base
A foo(B b)

Derived
C foo(D d)

Guidelines for inheritance

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

C <: A

Base
A foo(B b)

Derived
C foo(D d)

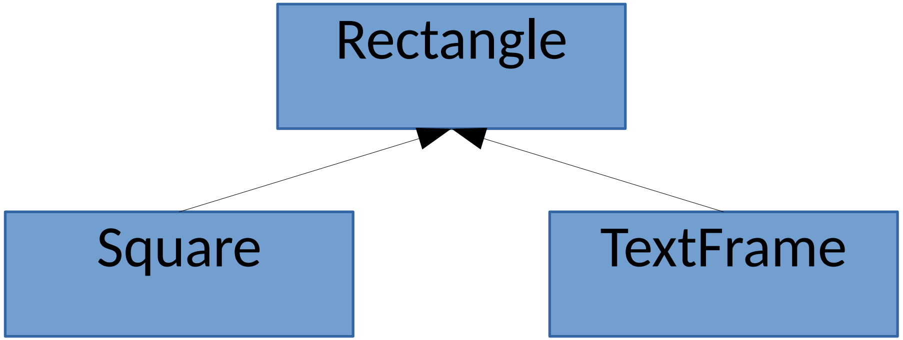
Return types are ***covariant***

Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - If φ is true for *a sequence of operations on* the base,
      then φ is true for *a sequence of operations on* the derived

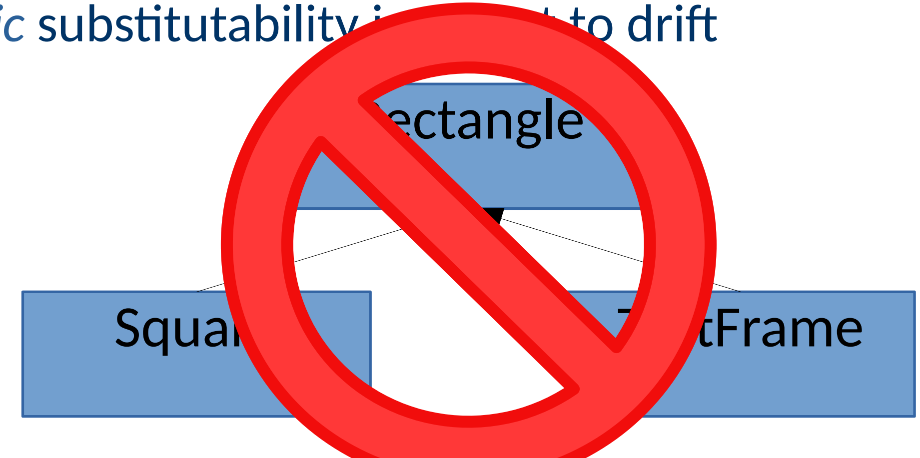
Guidelines for inheritance

- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - If φ is true for a sequence of operations on the base,
      then φ is true for a sequence of operations on the derived
  - *Semantic* substitutability is robust to drift



Guidelines for inheritance

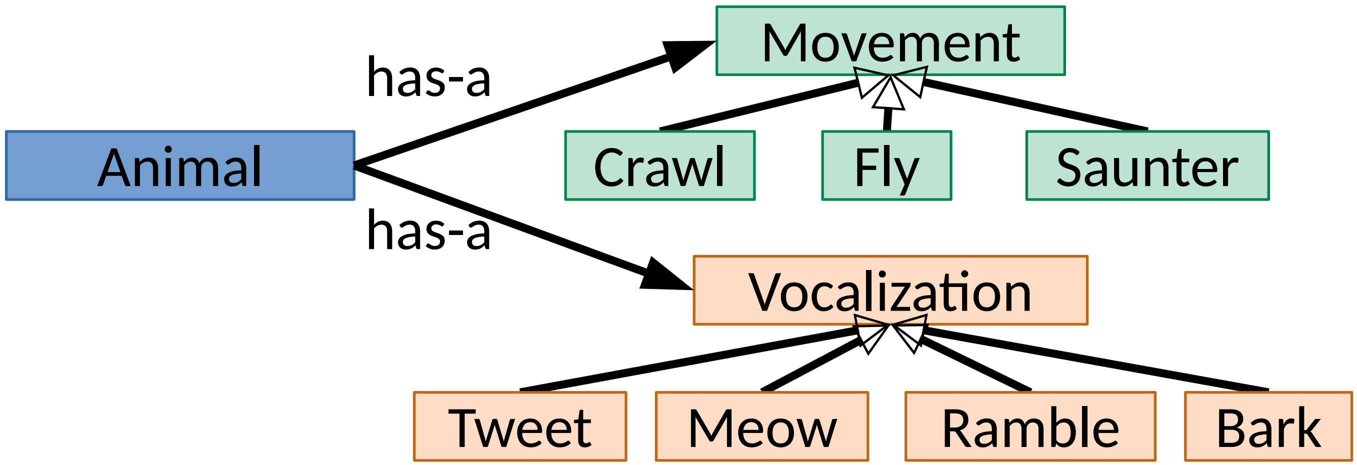
- Use inheritance for *semantic is-a* relationships
  - Liskov substitutability
    - If φ is true for the base, then φ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - If φ is true for a sequence of operations on the base, then φ is true for a sequence of operations on the derived
  - *Semantic* substitutability i … 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

[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

Alternatively: Only leaves should be instantiable.

[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

```
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

```
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

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

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

[Bloch, “Effective Java”]

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

Guidelines for inheritance

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

```
class Parent {
```

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

```
};class Child : public Parent {
public:
                                    }
};
```

Other patterns help even more…

[Bloch, "Effective Java"]

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

Guidelines for inheritance

- Design for inheritance.
  Choose *customization points* for runtime polymorphism.
  Prevent inheritance elsewhere.

```
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.