

Classes

Derived Classes and Inheritance

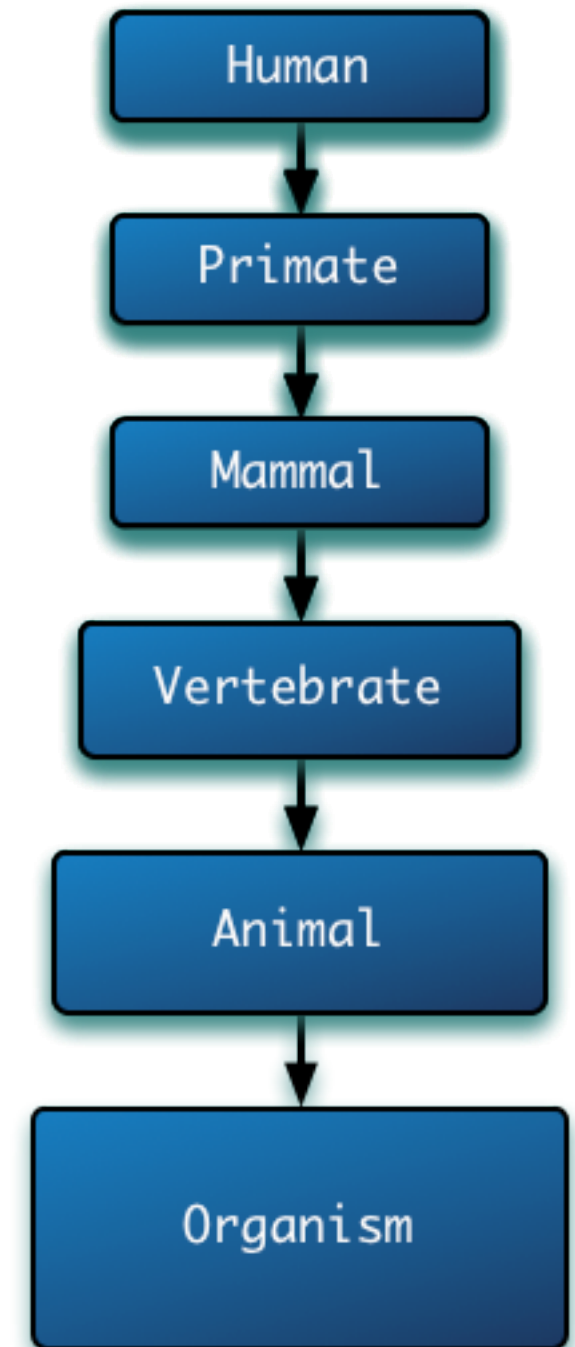
Humans

A Human...

- is a primate,
- which is a mammal,
- which is a vertebrate,
- which is an animal,
- which is an organism


This is a natural way to describe relations

- not surprisingly, most programming ways support some form of inheritance of classes
- inheritance is a powerful tool!



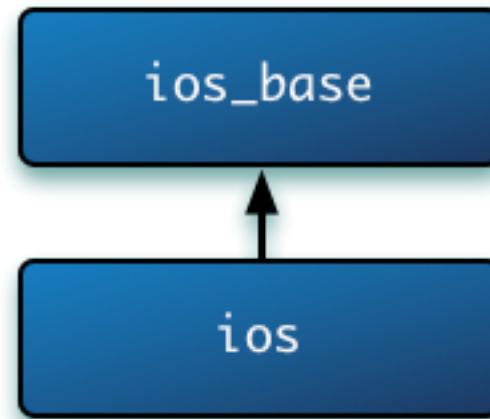
Streams are a great example...

Stream Class Hierarchies

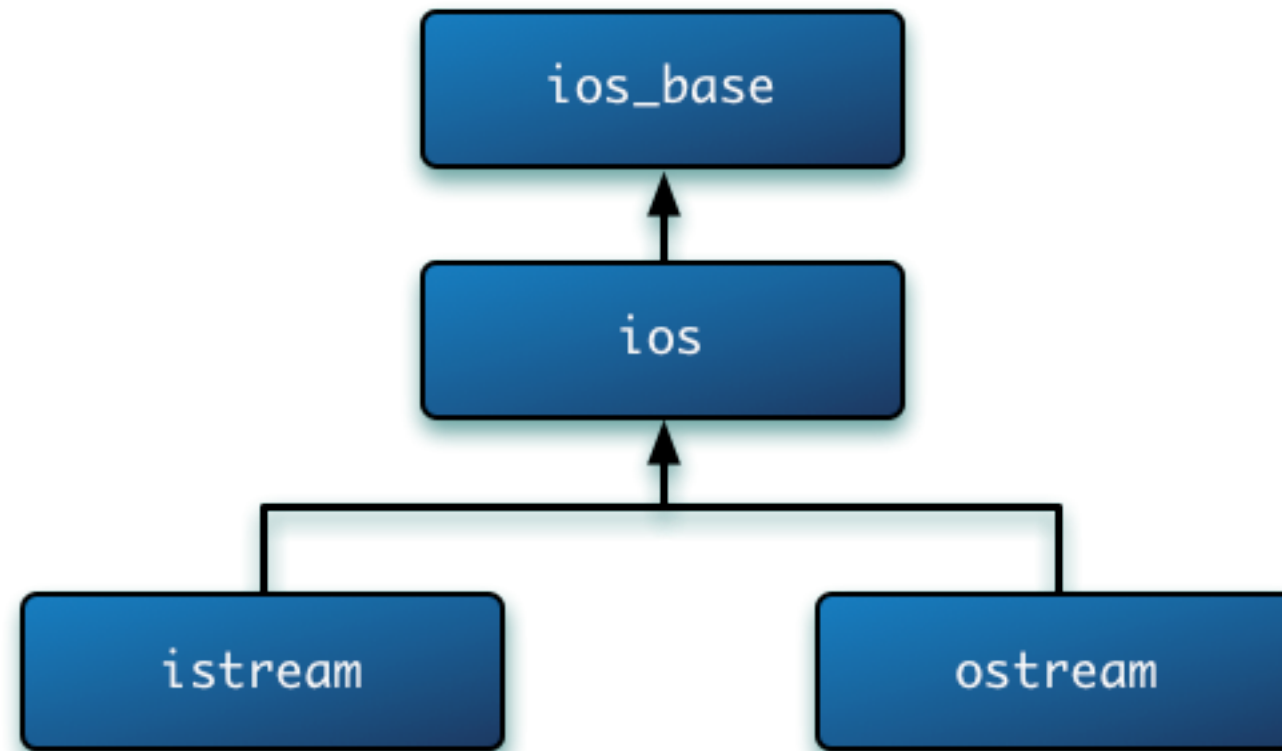


ios_base

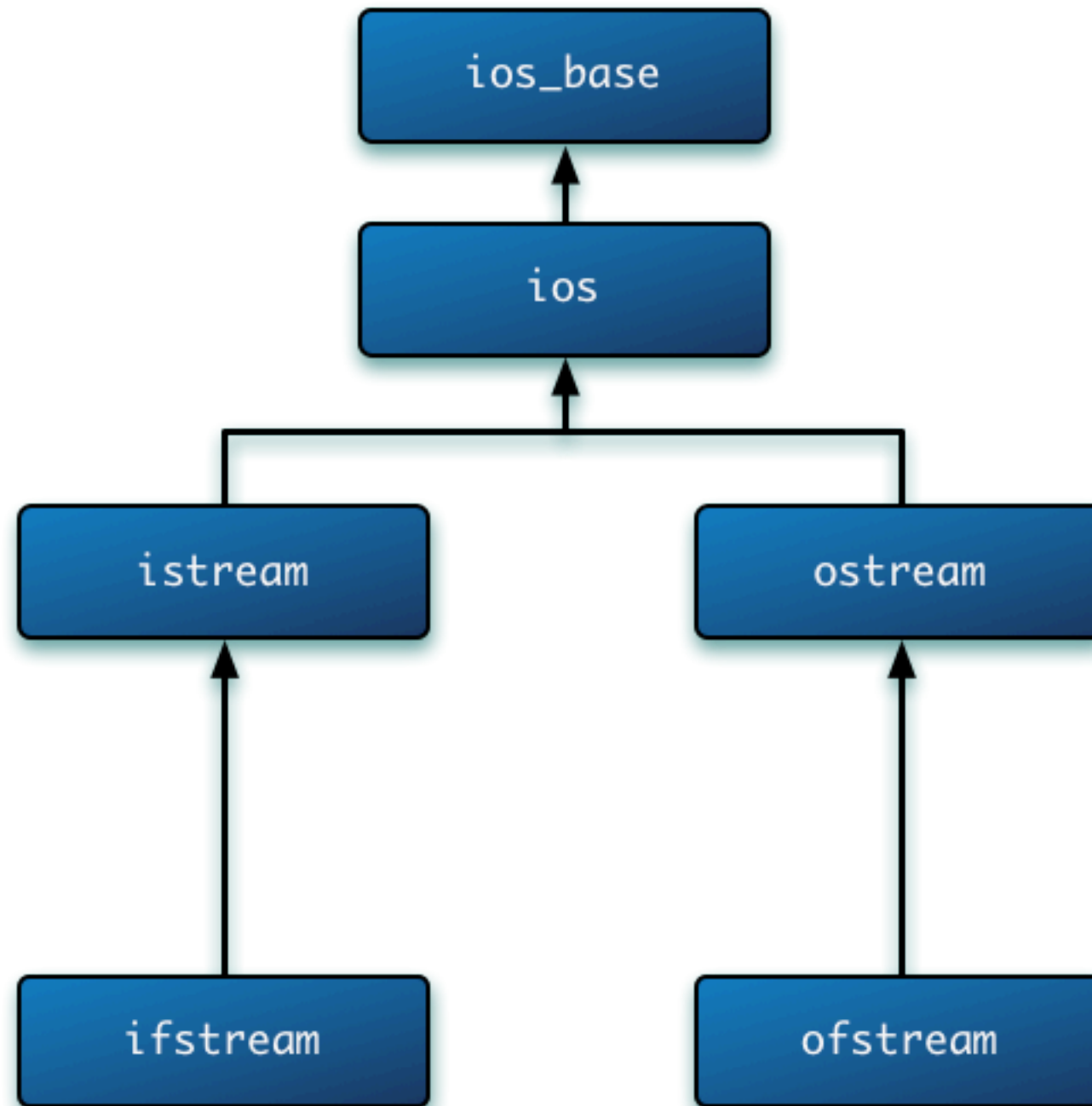
Stream Class Hierarchies



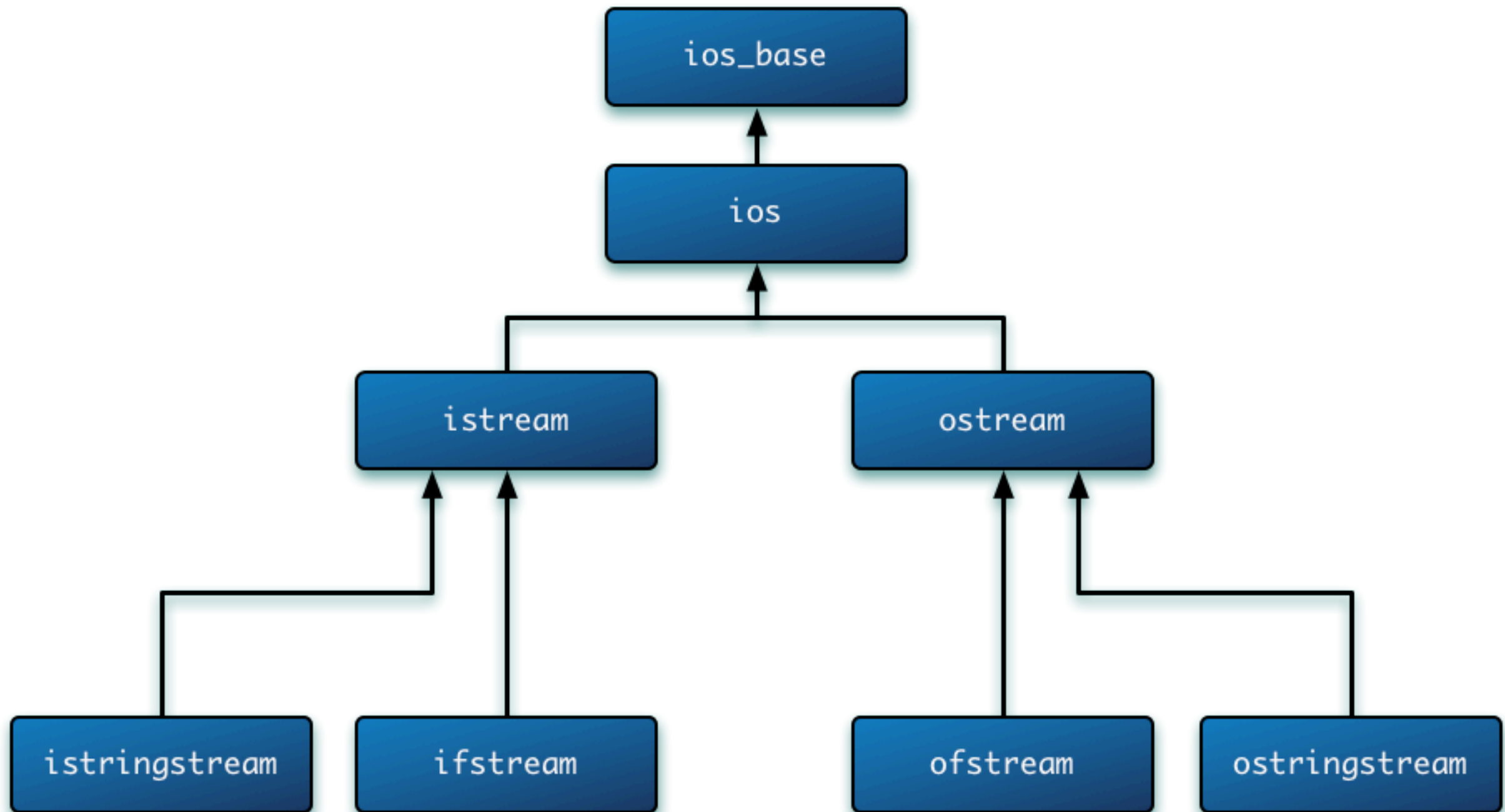
Stream Class Hierarchies



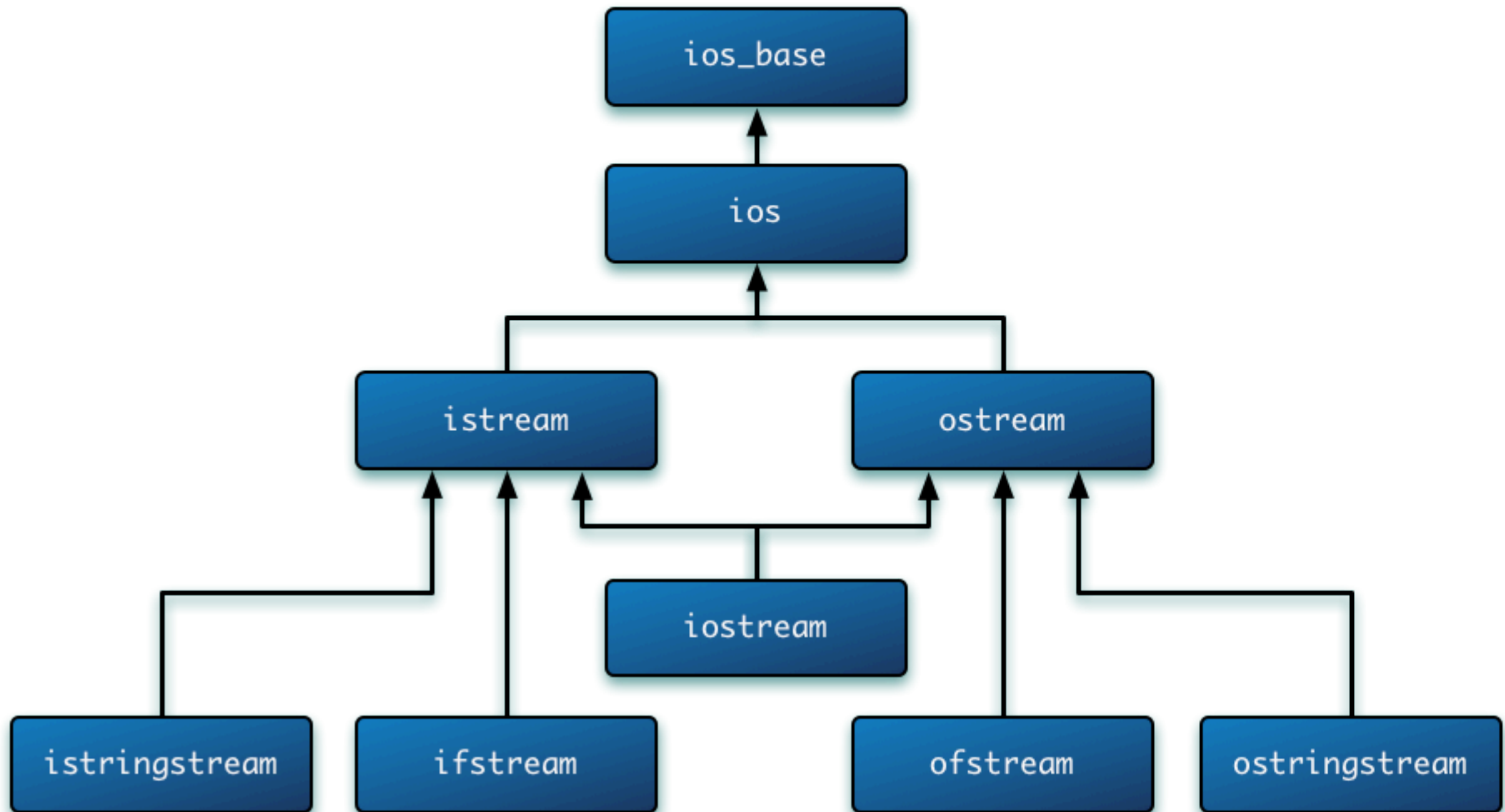
Stream Class Hierarchies



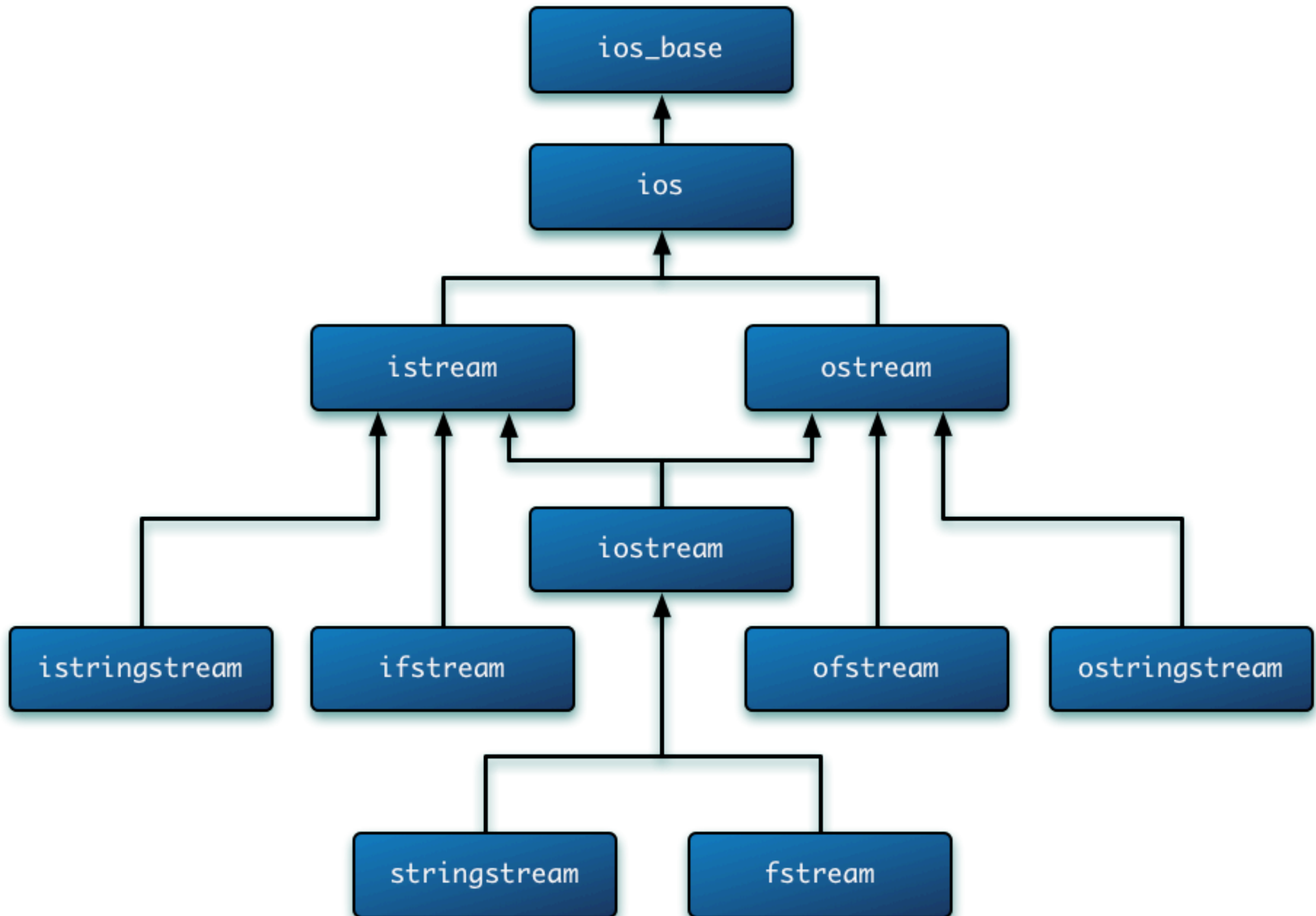
Stream Class Hierarchies



Stream Class Hierarchies



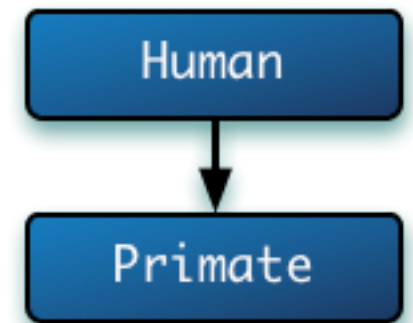
Stream Class Hierarchies



Inheritance

Class inheritance:

- enables highly reusable code
- effectively represents “is-a” relationships (e.g., a human is a primate)



Some terminology:

- base class: the parent class from which another inherits functionality
- derived class (also called subclass or child class): the class inheriting the behavior of the base class

The derived class will inherit everything from its parent!

- this includes data members, methods, constructors... everything!

Inheritance

Multiple classes can be derived from a single class

- all shared behavior should be put in the base class
- subclasses represent specialized versions of the base class, and only need to define the behavior and data members that their parent doesn't provide

Example:

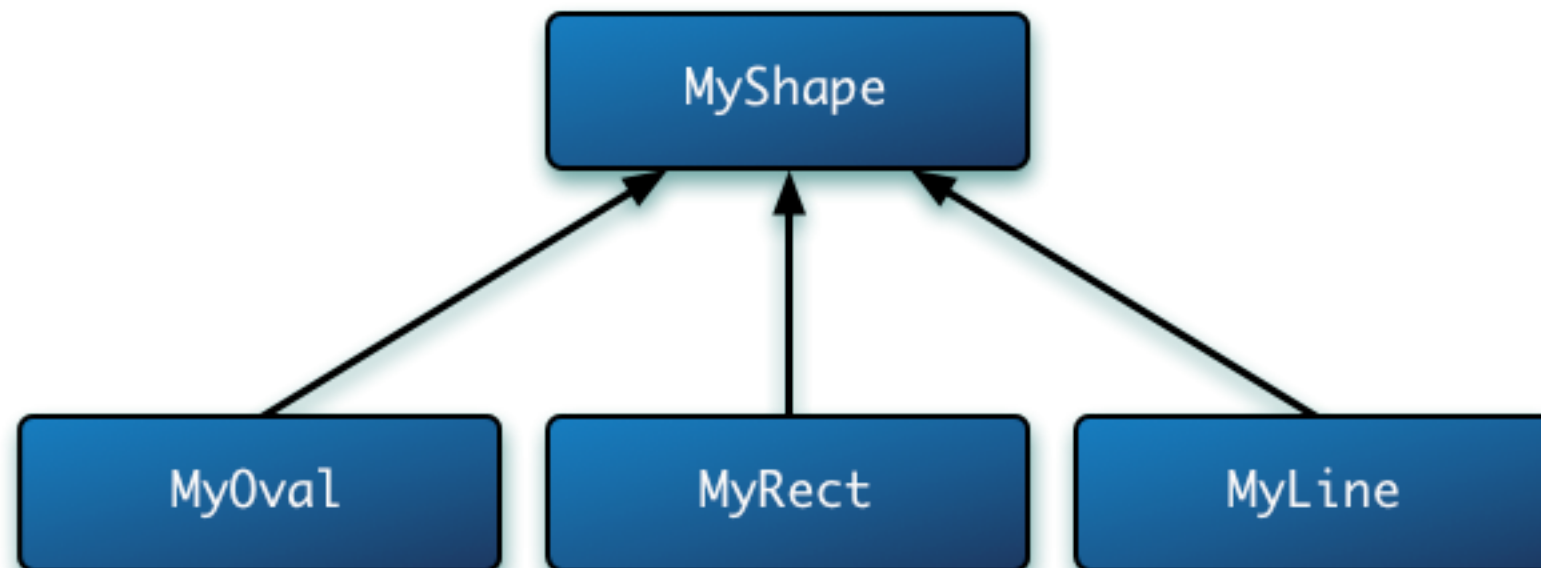
- assume that multiple shape classes (MyRect, MyOval and MyLine) all share similar properties and behaviors
- for instance, each class:
 - represents its location using a pair of points
 - has a color
 - should be able to draw itself
- these classes differ only slightly from one another, so...
- instead of redefining these shared properties in each class (not D.R.Y!), put them in a base class (MyShape) and have them all derive from it

Inheritance

Multiple classes can be derived from a single class

- all shared behavior should be put in the base class
- subclasses represent specialized versions of the base class, and only need to define the behavior and data members that their parent doesn't provide

Example:



Anything declared in MyShape will be part of the derived classes, too!

Each subclass will implement their own slightly different methods and properties

Inheritance

Assume we have the following class declared:

```
// the base class
```

```
class MyShape {
```

```
    // code
```

```
};
```

Inheritance

We can inherit from the class in two different ways:

// class that derives from MyShape (public base class)

```
class MyRect : public MyShape {
```

```
    // code
```

```
};
```

// class that derives from MyShape (private base class)

```
class MyRect : private MyShape {
```

```
    // code
```

```
};
```

public vs private Base Classes

Both of these classes derive from MyShape:

```
class MyRect : public MyShape { };
```

```
class MyRect : private MyShape { };
```

A **public** base class:

- makes the **public** members of the base class available as **public** members of the derived class

A **private** base class:

- makes the **public** members of the base class available as **private** members of the derived class

In both cases, all the members of MyShape are part of MyRect!

public vs private Base Classes

Example of a **public** base class:

```
class MyShape {  
    public:  
        MyShape* clone() const;  
    private:  
        Color color;  
};  
class MyRect : public MyShape { };
```

The MyRect class inherits all members of MyShape (clone and color)

- it cannot directly access color (a **private** property of the MyShape class)
- clone (a **public** method of the MyShape class) acts just like a **public** method of MyRect

public vs private Base Classes

Example of a **private** base class:

```
class MyShape {  
    public:  
        MyShape* clone() const;  
    private:  
        Color color;  
};  
class MyRect : private MyShape { };
```

The MyRect class inherits all members of MyShape (clone and color)

- it cannot directly access color (a **private** property of the MyShape class)
- clone (a **public** method of the MyShape class) acts just like a **private** method of MyRect

protected

You should be familiar with **public** and **private**:

- **private** members are only visible to the class that defines them (not to derived classes or other functions)
- **public** members are visible everywhere (derived classes included)
- in a derived class, **public** members of the base class are treated as:
 - **public** members of the derived class when the base class is **public**
 - **private** members of the derived class when the base class is **private**

There's a third visibility level: **protected**

- **protected** members are visible both to the class that defines them AND to any derived classes
- this is useful when a base class needs to expose properties to derived classes but still prevent the world at large from seeing them

protected

Example class with different visibilities:

```
class MyShape {  
    public:    // visible everywhere  
        MyShape* clone() const;  
    protected: // visible to this class & derived classes  
        Color color;  
    private:   // visible only to this class  
        PlansForWorldDomination plans;  
};
```

Inheritance Example

Let's say we have the following class:

```
class Clock {  
    public:  
        Clock();  
        void set_time(int hour, int min, bool morning);  
        void advance(int mins);  
        int get_hour() const;  
        int get_minute() const;  
        bool is_morning() const;  
    private:  
        // appropriate properties  
};
```

Inheritance Example

Then we decided to make a CuckooClock class, too

```
class CuckooClock {  
    public:  
        CuckooClock();  
        void set_time(int hour, int min, bool morning);  
        void advance(int mins);  
        int get_hour() const;  
        int get_minute() const;  
        bool is_morning() const;  
        bool is_cuckooing() const;  
        // ...  
};
```

Inheritance Example

The CuckooClock adds a single new method beyond a regular clock...

```
bool is_cuckooing() const;
```

The rest of its logic is exactly the same...

- nasty duplication of code! ick....
- I think you see where this is going...

If we made CuckooClock a subclass of Clock, the duplication vanishes!

```
class CuckooClock : public Clock {  
    public:  
        bool is_cuckooing() const;  
};
```

Inheritance Example

The CuckooClock adds a single new method beyond a regular clock...

```
bool is_cuckooing() const;
```

The implementation for this method:

```
bool CuckooClock::is_cuckooing() const {  
    // call the get_minute method of the base class  
    return (get_minute() == 0);  
}
```

Everything in the Clock class is part of the CuckooClock class

- we can directly use `public` and `protected` members of the base class
- `private` properties of the base class are part of the derived class, but not directly accessible

Overriding Methods

Sometimes a method defined in the base class isn't quite enough...

You can override methods the method in the derived class!

- this lets you specialize methods to suit the needs of the derived class
- you can still access the original version inside the overridden version by prefixing the name of the function with the name of the base class and the scope-resolution operator

Overriding Methods

Let's say we needed a clock that uses military time....

- it could derive from the Clock class like before
- however, it needs to override the default get_hour behavior to return 0..23

Example:

```
class MilitaryClock : public Clock {  
    public:  
        // override get_hour method of the base class  
        int get_hour() const;  
};
```

The get_hour function was originally declared in the base class

Overriding Methods

Example implementation of overridden method:

```
int MilitaryClock::get_hour() const {  
    // call the base class' version of the method  
    int hour = Clock::get_hour();  
  
    // use another method defined by the base class  
    if (is_morning()) {  
        return (hour == 12) ? 0 : hour;  
    } else {  
        return (hour == 12) ? 12 : hour + 12;  
    }  
}
```

Using a Derived Class

Given the previous class definitions, we can do the following:

```
// declare some clocks
```

```
Clock regular_clock;
```

```
CuckooClock annoying_clock;
```

```
MilitaryClock military_clock;
```

```
// set the times (each shares base functionality)
```

```
regular_clock.set_time(3, 0, false); // 3 pm
```

```
annoying_clock.set_time(3, 0, false); // 3 pm
```

```
military_clock.set_time(3, 0, false); // 3 pm
```

Using a Derived Class

Given the previous class definitions, we can do the following:

```
// use the specialized cuckoo clock
```

```
annoying_clock.advance( rand() );
```

```
if (annoying_clock.is_cuckooing()) {
```

```
    cout << "SHUT UP!!!" << endl;
```

```
}
```

```
// use the normal and overridden version of get_hour
```

```
cout << regular_clock.get_hour() << endl; // 3
```

```
cout << military_clock.get_hour() << endl; // 15
```

Using a Derived Class

Objects of a derived class can be used as objects of the base class:

```
// returns true if LHS clock is earlier than RHS clock
```

```
bool operator <(const Clock& LHS, const Clock& RHS);
```

We can call this method using any Clock or derived class object!

```
if (regular_clock < annoying_clock) { /* code */ }
```

```
if (annoying_clock < military_clock) { /* code */ }
```

Using a Derived Class

Objects of a derived class can be used as objects of the base class:

```
// assigns derived class object to a base class object
```

```
regular_clock = annoying_clock; // works!
```

```
regular_clock = military_clock; // works!
```

A base class object **CANNOT** be used as a derived class object:

```
annoying_clock = regular_clock; // compiler error!
```

```
military_clock = regular_clock; // compiler error!
```

Just remember that an object can become less specialized

- but it can NEVER magically become more specialized

Automatic Constructors

C++ can provide an automatic default constructor:

- if you don't declare ANY constructors yourself, C++ gives you a default constructor that does two things...
- first, it activates the default constructor for the base class (to initialize any member variables that the base class uses)
- second, it activates the default constructor for any new member variables that the derived class has but the base class does not

It also provides an automatic copy constructor:

- if you don't explicitly declare a copy constructor, you get a default copy constructor similar to the automatic default constructor...
- first, it activates the copy constructor for the base class (to copy any member variables that the base class uses)
- second, it activates the copy constructor for any new member variables that the derived class has but the base class does not

Automatic Assignment Operator

C++ will also provides an automatic assignment operator:

- if you don't explicitly declare an assignment operator, you get a default implementation (surprise!)
- first, it activates the assignment operator for the base class (to copy any member variables that the base class uses)
- second, it activates the assignment operator for any new member variables that the derived class has but the base class does not

Detecting a pattern here?

Automatic destructor

Yep, C++ provides an automatic destructor, too!

- assuming you don't define your own, that is...
- first, it calls the destructor for member variables in the DERIVED class,
- THEN it calls the destructor for the base class
- this happens in the reverse order of the automatic constructors and the assignment operator