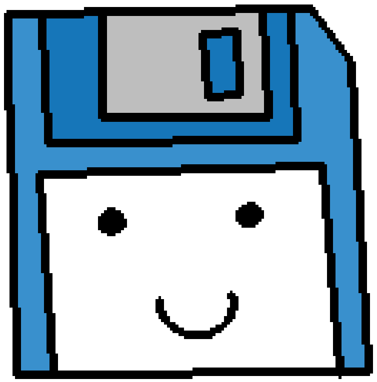
More with Classes: Inheritance

Written by Rachel J. Morris, last updated 2016-02-14

Reusing Functionality



Reusing Functionality

The idea behind inheritance is that it is a way to re-use code.

A lot of design we do in software development is for the sake of not duplicating code, if you think about it.

Let's say we have several classes with functionality that is common between them, but they're also different enough to need their own unique functions.

We could move those <u>shared</u> functions and variables into a single class, the **parent** class, and these classes now **inherit** the shared functionality from the parent.

Now these classes are **child classes**, and their unique functionality is seen as “specialization”, on top of the shared functionality they all have.

Reusing Functionality

For example, let's say that we're designing a user interface. What objects would we need?

Button

Click Me

Label

Label

Link

Go Elseware

Image

Checkbox

Is true?

List

- Item 1
- Item 2
- Item 3

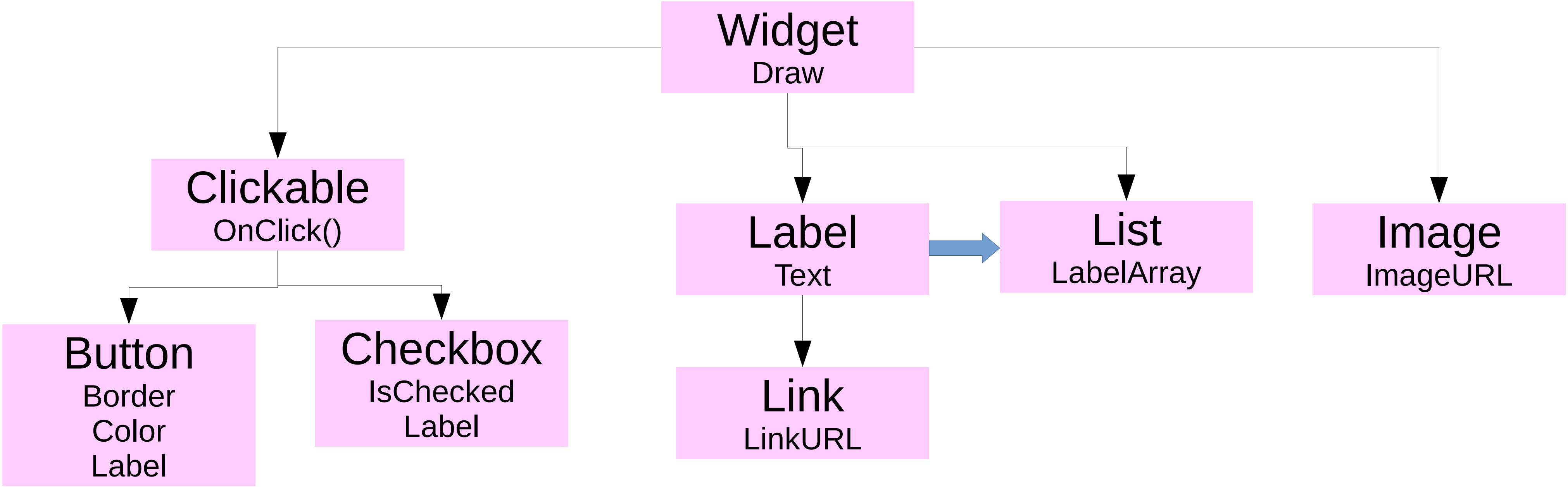
Some of these might have **text** in common, or they would have a behavior that is run when clicked.

Some of these might contain other classes. For example, a **List** object might contain an array of **Labels**.

All of these need functionality that **Draw** them to the screen.

Sometimes, you'll see a generic User Interface object called a **Widget.**

Reusing Functionality



A widget family might look something like this – A series of **parents and children**, through inheritance.

Widget would be the great grandparent that every other User Interface object would eventually be derived from.

Reusing Functionality

List
Labels[100]

Label
Text

ImageButton
Image
Button

Image
ImageURL

Button
Border
Color
Label

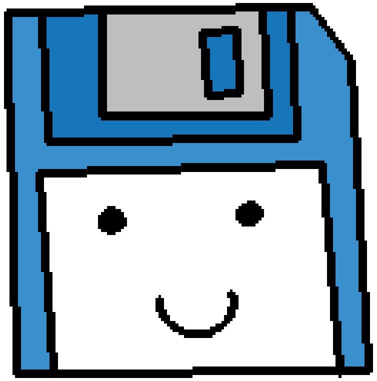
Label
Text

Then you could expand more, by having classes contain others...

Button contains a Label,

and an ImageButton contains an Image and a Button.

Inheritance in the Code



Inheritance in the Code

First, we need to create a **base class**, which is just a normal class.

```
class Animal
{
    public:
    void SetName( const string& name )
    {
        m_name = name;
    }

    void Speak()
    {
        cout << m_name << ": Huh?" << endl;
    }

    protected:
    string m_name;
};
```

When using inheritance, any **member variables** that you want the child classes to use, should be set as **protected**, instead of **private**. More on this later.

Inheritance in the Code

Then, to have another class inherit, you declare the class and put
: public PARENTCLASSNAME
on the same line as the **class**

```
class Cat : public Animal
{
    public:
    void Speak()    // Overwrite Function
    {
        cout << m_name << ": MEOW!" << endl;
    }

    void Claw()     // New functionality
    {
        cout << m_name
            << " destroys all your furniture" << endl;
    }
};
```

Inheritance in the Code

```
class Animal
{
    public:
    void SetName( const string& name );
    void Speak();

    protected:
    string m_name;
};

class Cat : public Animal
{
    public:
    void Speak();      // Overwrite Function
    void Claw();       // New functionality
};
```

Now our **child class** will have:

- Functionality inherited from the parent – **`SetName(...)`**
- Functionality from the parent that it is overwriting – **`Speak()`**
- Its own unique functionality – **`Claw()`**

Even though we don't see
`void SetName(...)`
In the Cat class declaration,
Because it was **public** in the parent class, it is automatically a part of this new class.

Inheritance in the Code

```
int main()
{
    Animal myAnimal;
    Cat myCat;

    myAnimal.SetName( "Bessie" );
    myCat.SetName( "Fluffy" );

    myAnimal.Speak();
    myCat.Speak();

    myCat.Claw();

    return 0;
}
```

shared

overwritten

unique to Cat

Now we can declare classes of either type, and utilize its shared functions in the same way.

Only the child can use functions that were uniquely declared within the child class.

Simple Inheritance

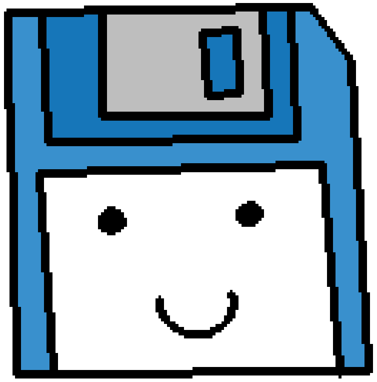
```
Bessie: Huh?
Fluffy: MEOW!
Fluffy destroys all your furniture
```

Inheritance in the Code

```
class Animal
{
    public:
    void SetName( const string& name )
    {
        m_name = name;
    }

    void Speak()
    {
        cout << m_name << ": Huh?" << endl;
    }

    protected:
    string m_name;
};
```

But what is this “protected” stuff?

Public, Private, & Protected



Public, Private, & Protected

We didn't need the **protected** keyword before, without inheritance.
We were only using **public** and **private** to restrict access to certain parts of our classes.

If you set a member variable or function to **private**,
then any **child classes** <u>cannot</u> access those private members.

There might be times when you do not want functionality to be shared with **child classes**,
so using **private** here would be appropriate.

However, if you set a member variable or function to **protected**, it is still treating it like a private member (nothing outside the class can access it).

However when you have a **child class**, it also becomes a protected member of the child class – so, private to any non-family,
but available to descendants.

Public, Private, & Protected

```
class Document
{
    public:
    void Output( const string& filename );
    void SetText( const string& text );

    protected:
    string m_text;

    private:
    // Unique function
    void WriteTextFile( const string& filename );
};
```

```
class WebDocument : public Document
{
    public:
    // overwrite Output
    void Output( const string& filename );

    protected:

    private:
    // Unique function
    void WriteHtmlFile( const string& filename );
};
```

Here, we have two Document classes – one saves plaintext (.txt), and the other saves a web (.html) document.

Therefore, they both have **private** functions for **`WriteTextFile`** and **`WriteHtmlFile`**.
These aren't shared, and are unique to each class.

Instead, they have a shared **`Output`** function,
which takes in the file name, then calls the Write function.

Public, Private, & Protected

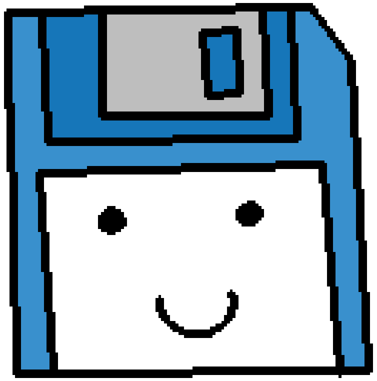
```
class Document
{
    public:
    void Output( const string& filename );
    void SetText( const string& text );

    protected:
    string m_text;

    private:
    // Unique function
    void WriteTextFile( const string& filename );
};
```

```
class WebDocument : public Document
{
    public:
    // overwrite Output
    void Output( const string& filename );

    protected:

    private:
    // Unique function
    void WriteHtmlFile( const string& filename );
};
```

To the user / other programmers, they have the same interface, but behind-the-scenes, they work differently.

```
int main()
{
    Document doc;
    WebDocument webDoc;

    doc.SetText( "Hello, World!" );
    webDoc.SetText( "Hello, World!" );

    doc.Output( "TextFile.txt" );
    webDoc.Output( "WebFile.html" );

    return 0;
}
```

Function Redefining and Function Overloading



Redefine & Overload

We have had examples of **overwriting** (or **redefining**) member functions.

This is when, in our **child** class, we redeclare a function that the **parent** has. The redeclaration has the same **return type**, **name**, and **parameter list**.

```
void Document::Output( const string& filename )
{
    WriteTextFile( filename );
}

void WebDocument::Output( const string& filename )
{
    WriteHtmlFile( filename );
}

int main()
{
    Document doc;
    WebDocument webDoc;

    doc.Output( "TextFile.txt" );
    webDoc.Output( "WebFile.html" );

    return 0;
}
```

This isn't the same as function overloading.

The function is called in the same way, but will go to the special **child** version instead of the **parent** version.

Redefine & Overload

Even if we **redeclare** a function, we can still call the parents' version.

We might do this if the new function expands on top of the old functionality, and we don't want to duplicate the old code.

```cpp
class Person
{
    public:
    void Setup( const string& name )
    {
        m_name = name;
        m_location = "unset";
    }

    protected:
    string m_name;
    string m_location;
};
```

```cpp
class Student : public Person
{
    public:
    void Setup( const string& name )
    {
        Person::Setup( name );
        m_gpa = 0;
    }

    protected:
    float m_gpa;
};
```

The **parent** version of **`Setup`** works just fine, we just need to add some more initialization afterwards, so call the parent.

Redefine & Overload

To call the parents' version of the function from within the **overwritten / redefined** function, use the parent class name, followed by the scope resolution operator :: , and then the function name with any arguments needed.

```
class Person
{
    public:
    void Setup( const string& name )
    {
        m_name = name;
        m_location = "unset";
    }

    protected:
    string m_name;
    string m_location;
};
```

```
class Student : public Person
{
    public:
    void Setup( const string& name )
    {
        Person::Setup( name );
        m_gpa = 0;
    }

    protected:
    float m_gpa;
};
```

Redefine & Overload

We could also **overload** parent functions.
Remember that **overloading** is using the same function name,
but a different parameter list.

```cpp
class Person
{
    public:
    void Setup( const string& name )
    {
        m_name = name;
        m_location = "unset";
    }

    protected:
    string m_name;
    string m_location;
};
```

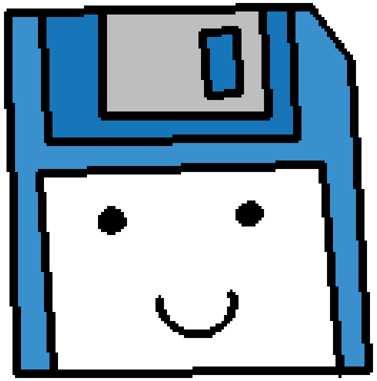
We can also still call a parent function as-needed.

```cpp
class Student : public Person
{
    public:
    void Setup( const string& name )
    {
        Person::Setup( name );
        m_gpa = 0;
    }

    void Setup( const string& name, float gpa )
    {
        Person::Setup( name );
        m_gpa = gpa;
    }

    protected:
    float m_gpa;
};
```

overwritten/
redefined

overloaded

More Constructors



Constructors & Inheritance

Similar to calling the **base class'** functions, we can also call the base class' constructor.

However, to call a parent **constructor**,
we must do so through the **initializer list**.

An initializer list is another way to initialize member variables in a class, rather than manually setting everything within the constructor body.

Constructors & Inheritance

No initializer list

```
class Person
{
    public:
    Person( const string& name )
    {
        m_name = name;
        m_location = "unset";
    }

    protected:
    string m_name;
    string m_location;
};
```

With initializer list

```
class Person
{
    public:
    Person( const string& name ) : m_name( name ), m_location( "unset" )
    {
    }

    protected:
    string m_name;
    string m_location;
};
```

Constructors & Inheritance

```
Person( const string& name ) : m_name( name ), m_location( "unset" )
{
    cout << "Person constructor" << endl;
}
```

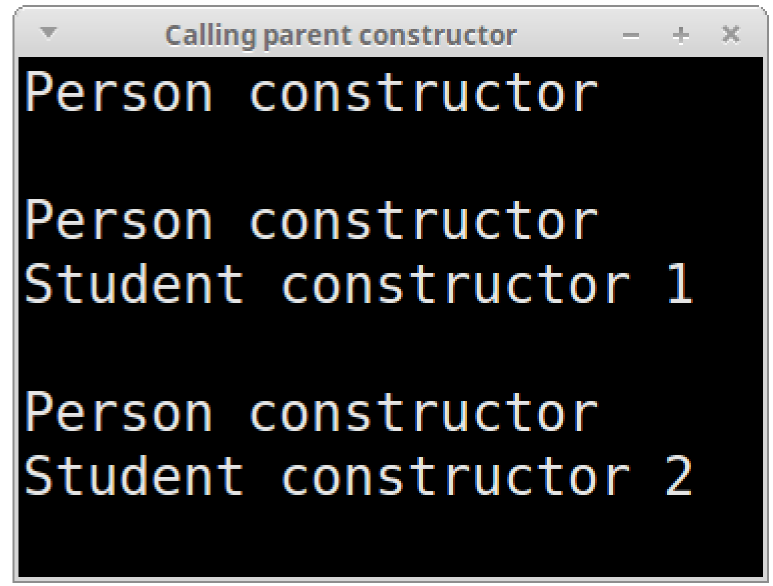
In order to call a parent constructor, we need to do so within an **initializer list**, like the following:

```
class Student : public Person
{
    public:
    Student( const string& name ) : Person( name ), m_gpa( 0 )
    {
        cout << "Student constructor 1" << endl;
    }

    Student( const string& name, float gpa ) : Person( name ), m_gpa( gpa )
    {
        cout << "Student constructor 2" << endl;
    }

    protected:
    float m_gpa;
};
```

Constructors & Inheritance

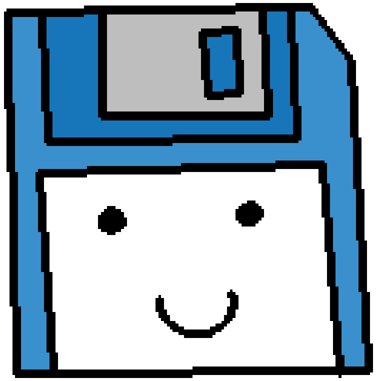
Then, when we create instances of our classes:

```cpp
Person person( "Ada" );
cout << endl;
Student studentA( "Stephanie" );
cout << endl;
Student studentB( "Grace", 4.0 );
```



Notice that the parent constructor will be called first, before the student constructor.

Additional Notes



: public BaseClass

You might have noticed that we use the **public** keyword when we are inheriting from another class, but what happens if we use **protected** or **private**?

```
class Student : public Person

class Student : protected Person

class Student : private Person
```

For the most part, you will be using public inheritance.

With **private** inheritance,
any **public** or **protected** members of the parent class now become **private** members of the child class.

With **protected** inheritance,
any **public** or **protected** members of the parent class now become **protected** members of the child class.

These are used for more advanced design, so don't worry about it.

Multiple Inheritance

Classes can also inherit from multiple parent classes.

Let's say that we have a “**Button**” class and an “**Image**” class, we could use the functionality and variables from both classes to make an “**ImageButton**” class, by inheriting from both.

Any protected and public members from **both classes** now become protected / public members of the **child class**, merging the two parents.

Multiple Inheritance

```cpp
class Button
{
    public:
    void Draw();
    void SetDimensions( int width, int height );

    protected:
    int m_width, m_height;
};
```

```cpp
class Label
{
    public:
    void Draw();
    void SetText( const string& text );

    protected:
    string m_text;
};
```

```cpp
class TextButton : public Button, public Label
{
    public:
    void Draw();

    // inherits m_text, m_width, m_height
    // and the SetText and SetDimensions functions
    // Overwrites Draw function
};
```

Now the child class will have any **non-private members** of both of its parents.

```
Label label;
label.SetText( "This is a label" );

Button button;
button.SetDimensions( 10, 5 );

TextButton textButton;
textButton.SetText( "Click Me" );
textButton.SetDimensions( 10, 5 );

cout << endl << endl << "LABEL" << endl;
label.Draw();

cout << endl << endl << "BUTTON" << endl;
button.Draw();

cout << endl << endl << "TEXT BUTTON" << endl;
textButton.Draw();
```

Multiple Inheritance

```


LABEL
This is a label


BUTTON
##########
#        #
#        #
#        #
##########


TEXT BUTTON
##########
#        #
#Click Me#
#        #
##########
```

Voilà!

But, inheriting from too many parents can cause the internal code to become messy, rather than concise and compartmentalized...

Is-A, Has-A

Utilizing **inheritance** is usually considered an “Is-A” relationship.

This means that, if **TextButton** inherits from **Label** and **Button**,
what we're saying is that
“TextButton Is A Label and Is A Button”.

There's another design style that is considered a “Has-A” relationship: Composition.

Is-A, Has-A

Utilizing composition is essentially making one class a **member** of another class, rather than a **parent or child**.

```
class TextButton : public Button, public Label
{
    public:
    void Draw();

    // inherits m_text, m_width, m_height
    // and the SetText and SetDimensions functions
    // Overwrites Draw function
};
```

INHERITANCE:
IS-A

The TextButton **is a child of** both the Button and Label classes.

```
class TextButton
{
    public:
    void Draw();
    void SetText( const string& text );
    void SetDimensions( int width, int height );

    private:
    Button m_button;
    Label m_label;
};
```

COMPOSITION:
HAS-A

The TextButton **contains** Button and Label as members.

Is-A, Has-A

```
class TextButton
{
    public:
    void Draw();
    void SetText( const string& text );
    void SetDimensions( int width, int height );

    private:
    Button m_button;
    Label m_label;
};
```

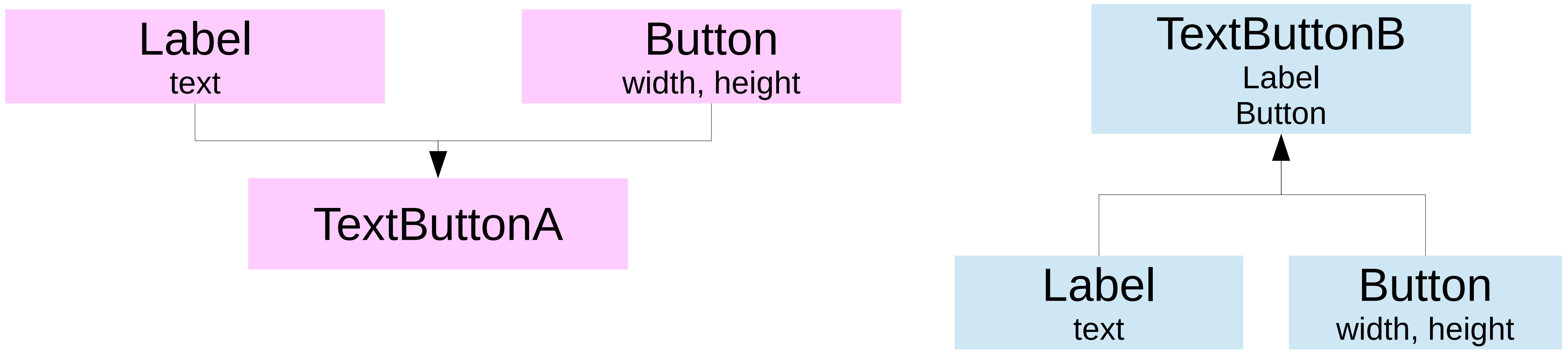
By making **Button** and **Label** members of the **TextButton** class instead of parents of it, we have nicely **compartmentalized** the “Button” functionality and “Label” functionality within the **`m_button`** and **`m_label`** objects.

```
void Button::SetDimensions( int width, int height )
{
    m_width = width;
    m_height = height;
}
```

```
void TextButton::SetDimensions( int width, int height )
{
    m_button.SetDimensions( width, height );
}
```

We are adding another **layer** to the object design.

Is-A, Has-A

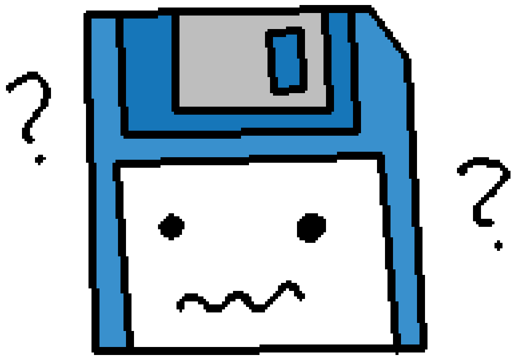


There are arguments to be made for designing your objects either way, so ultimately it is up to you.
Read other peoples' opinions, consider them, figure out what works for you.

That was a lot to cover!

You don't have to be a master of all these topics right now – we're mostly going to focus on vanilla inheritance,

but it is good to know what is possible, so if you encounter it later, you can do more research!



Let's work on an example of using inheritance.

Remember that there is also sample code for each of these topics in the course repository!