Flaws of Inheritance

"Prefer **composition** over inheritance"

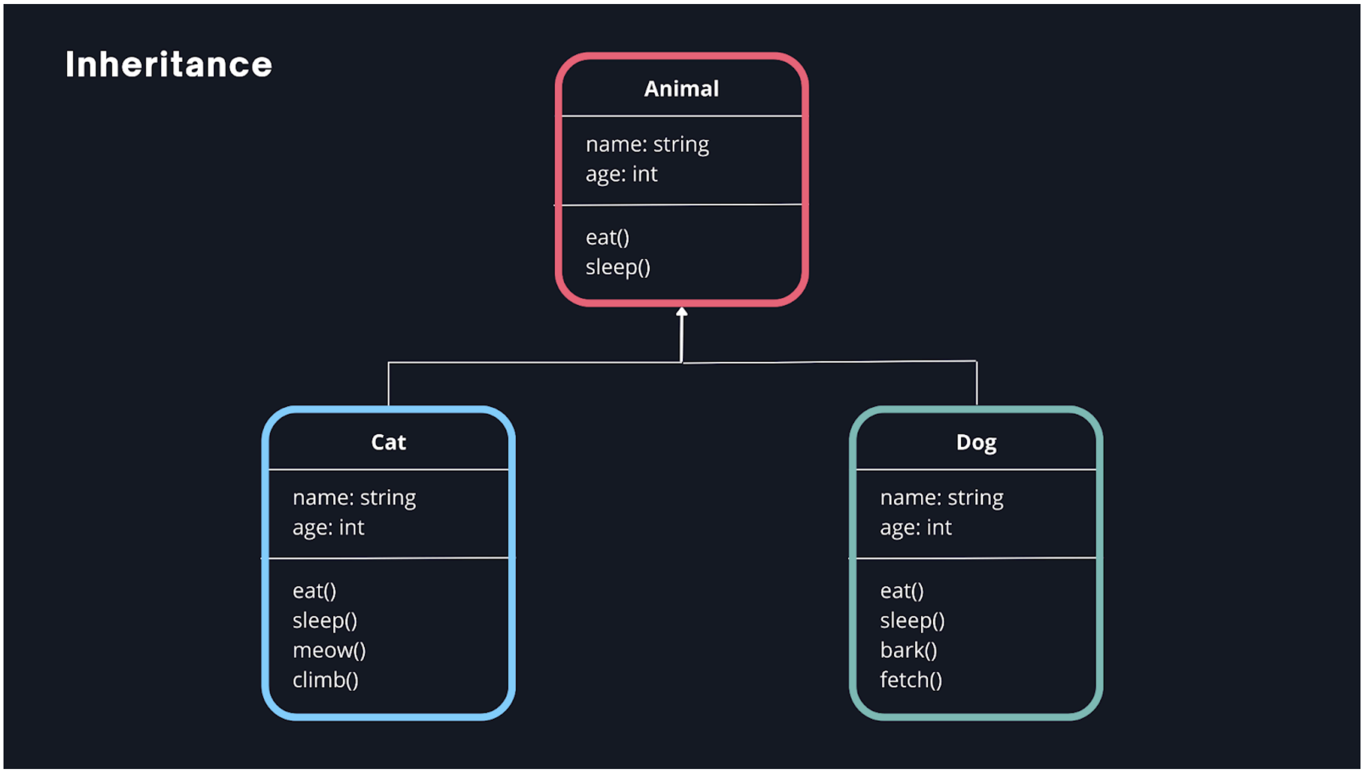
So what is composition? What is inheritance? And why would you care for one over the other?

Composition > Inheritance

Both composition and inheritance are trying to solve the same problem — You have a piece of code that you're trying to **reuse**.

Inheritance

Inheritance is when you reuse the functionality of a class by extending its functionality in a subclass.



When you inherit a class you can inject new methods in the child class to extend or override parts.

Let's look at our example `Image`

> Let's take a look at our parent class `Image`, which has multiple methods to alter images. Our program should be able to support `jpeg`, `png` and `bitmap` images.

```
abstract class Image {
    private Pixel[,] pixels = new Pixel[0,0];
    public int Width { get; privat set; }
    public int Height { get; private set; }

    public Image(int width, int height) {
        this.setSize(width, height);
    }

    public Image() : this(0,0) {}

    public void setSize(int width, int height) {
        this.Width = width;
        this.Height = height;
        this.pixels = new Pixel[height, width];
    }

    public Pixel pixelAt(int x, int y) {
        return pixels[y, x]
    }

    public ref Pixel this[int x, int y] {
        get {
            return ref pixels[y, x]
        }
    }

    public void resize(double scale) {
        // imagine code is here
    }

    public void flipHorizontal() {
        // imagine code is here
    }

    public void flipVeritcal() {
        // imagine code is here
    }

    public abstract void save();
    public abstract void load();
}
```

We want to also reuse these methods for all the different kinds of images we have: `jpeg`, `png`, and `bitmap`.

So let's add two `abstract` methods `save` and `load`!

Let's write these new subclasses `JpegImage`, `PngImage` and `BmpImage`.

- Each subclass implements it's very own version of `load` and `save`.
  - Recall that an abstract method cannot be implemented in the parent class. It must be implemented within a child class.
- But also get all of the other methods for free!
  - When we write a `JpegImage` we can call `jpeg.resize`!

```
class JpgImage : Image {
    private string path;
    private JpegOptions options;

    public JpgImage(string path, JpegOptions options) {
        this.path = path;
        this.options = options;
    }

    public override void save() {
        JpegEncoder encoder = new JpegEncoder(options);
        using (FileStream stream = File.OpenWrite(path)) {
            encoder.Encode<RGB>(this, stream);
        }
    }

    public override void load() {
        JpegDecoder decoder = new JpegDecoder();
        using (FileStream stream = File.OpenRead(path)) {
            Pixel[,] pixels = decoder.Decode<RGB>(options, stream);
            this.replacePixels(pixels);
            this.setSize(pixels.GetLength(1), pixels.GetLength(0));
        }
    }
}
```

```
class PngImage : Image {
    private string path;

    public PngImage(string path, PngOptions options) {
        this.path = path;
        this.options = options;
    }

    public override void save() {
        PngEncoder encoder = new PngEncoder(options);
        using (FileStream stream = File.OpenWrite(path)) {
            encoder.Encode<RGB>(this, stream);
        }
    }

    public override void load() {
        PngDecoder decoder = new PngDecoder();
        using (FileStream stream = File.OpenRead(path)) {
            Pixel[,] pixels = decoder.Decode<RGB>(options, stream);
            this.replacePixels(pixels);
            this.setSize(pixels.GetLength(1), pixels.GetLength(0));
        }
    }
}
```

```
class BmpImage : Image {
    private string path;

    public BmpImage(string path) {
        this.path = path;
    }

    public override void save() {
        System.Drawing.Bitmap bitmap = new System.Drawing.Bitmap(Width, Height)

        for (int row = 0; row < Height; row++) {
            for (int column = 0; column < Width; column++) {
                Pixel pixel = pixelAt(column, row);
                bitmap.SetPixel(column, row, Color.FromArgb(pixel.Red, pixel.Gr
            }
        }

        bitmap.Save(this.path);
    }

    public override void load() {
        System.Drawing.Bitmap bitmap = new System.Drawing.Bitmap(Width, Height)
        this.setSize(bitmap.Width, bitmap.Height);
        for (int row = 0; row < Height; row++) {
            for (int column = 0; column < Width; column++) {
                Color pixel = bitmap.GetPixel(column, row);
                this[row, column] = new Pixel(pixel.R, pixel.G, pixel.B)
            }
        }
    }
}
```

As we mentioned earlier, each of these classes get the concrete methods inherited from the parent class for free.

- This means `JpegImage` and `PngImage` can call the `resize` method.

```
JpgImage jpeg = new JpegImage("image.jpg", new JpegOptions);
jpeg.load();
jpeg.resize(0.5)
jpeg.save()


PngImage png = new PngImage("image.jpg", new PngOptions);
png.load();
png.resize(0.5)
png.save()
```

> The `resize` method is reused for ***all*** of the image types. But when we call the `load` and `save` methods, the overwritten version is invoked instead. This works well.

Problems with Inheritance

BUT, now we want to create a new type of `Image`, one that doesn't come from a file but rather from a drawable image. So we create the class `DrawableImage`.

This is where inheritance has issues!!!

The downsides of inheritance is that you have **coupled** yourself to the parent class.

- **The structure of the parent is forced upon the child.**
    - We are forced to implement the `save` and `load` methods in `DrawableImage` even though we just want to use our `resize`, `flipHorizontal` and `flipVertical` methods.

```
class DrawableImage : Image {
    private Pencil brush;
    public DrawableImage() {
        this.brush = new Pencil();
    }

    public void drawLine(int startX, int startY, int endX, int endY) { ... }

    public void drawPoint(int x, int y) {
        this.brush.drawPoint(x, y, this);
    }

    public override void save() {
        // there's an issue tho!!
        throw new InvalidOperationException("DrawableImage cannot save")
    }

    public override void load(){
        // there's an issue tho!!
        throw new InvalidOperationException("DrawableImage cannot load")
    }
}
```

The best we can do with our overrided methods is throw an exception error.

So what can we do now?

We can remove `save` and `load` and then put them in a new abstract class in between `Image` and `DrawableImage` named `FileImage`.

- BUT WAIT, this also breaks when anyone writes code that expects image to contain those methods.

```
abstract class FileImage : Image {
    public abstract void save();
    public abstract void load();
}

class ImageApp {
    Image image:
    void saveClicked() {
        image.save(); // this line will break!!!
    }
}
```

> A "small" change such as this can be a **costly refactor**. We then have to change all of our classes because we extracted `save` and `load` out of `Image` and into it's own class.

This is the biggest downfall of inheritance!!

Inheritance breaks down when you have to change your code.

"Change is the enemy of perfect design"

> **Inheritance forces us to predict the future** and structure the code accordingly. We humans are not good at predicting the future so we should be using inheritance sparingly.

You often will back yourself into a corner early on with your inheritance design.

- This is because **inheritance naturally asks you to bundle all common elements into a parent class.**
  - As soon as there is an exception then you end up needing to make large changes.

In an inherited relationship, the child and parent classes will have **tight coupling** between them.

Composition

Composition is the pattern of when you reuse code without inheritance. It's a type of relationship where a class is created by combining the other classes.

Let's use **composition** instead now in our `Image` class.

1. Remove the abstract methods from `Image`
2. Now `Image` is no longer an abstract class
3. In our child classes, we no longer inherit `Image`
   1. Now our `save` and `load` methods will **no longer override anything**. They will be standalone.
4. Those methods were accessing a ton of members from the parent class, we'll simply pass in the `Image` into the `save` and `load` methods.

`Image.cs`

```
class Image {
   private Pixel[,] pixels = new Pixel[0,0];
```

```
    public int Width { get; privat set; }
    public int Height { get; private set; }

    public Image(int width, int height) {
        this.setSize(width, height);
    }

    public Image() : this(0,0) {}

    public void setSize(int width, int height) {
        this.Width = width;
        this.Height = height;
        this.pixels = new Pixel[height, width];
    }

    public Pixel pixelAt(int x, int y) {
        return pixels[y, x]
    }

    public ref Pixel this[int x, int y] {
        get {
            return ref pixels[y, x]
        }
    }
}
```

`JpgImage.cs`

```
class JpgImage {
    private string path;
    private JpegOptions options;

    public JpgImage(string path, JpegOptions options) {
        this.path = path;
        this.options = options;
    }

    public override void save(Image image) { // we now pass in an Image
        JpegEncoder encoder = new JpegEncoder(options);
        using (FileStream stream = File.OpenWrite(path)) {
            encoder.Encode<RGB>(this, stream);
        }
    }
    public override void load(Image image) { // we now pass in an Image
        JpegDecoder decoder = new JpegDecoder();
        using (FileStream stream = File.OpenRead(path)) {
            Pixel[,] pixels = decoder.Decode<RGB>(options, stream);
            this.replacePixels(pixels);
            this.setSize(pixels.GetLength(1), pixels.GetLength(0));
        }
    }
}
```

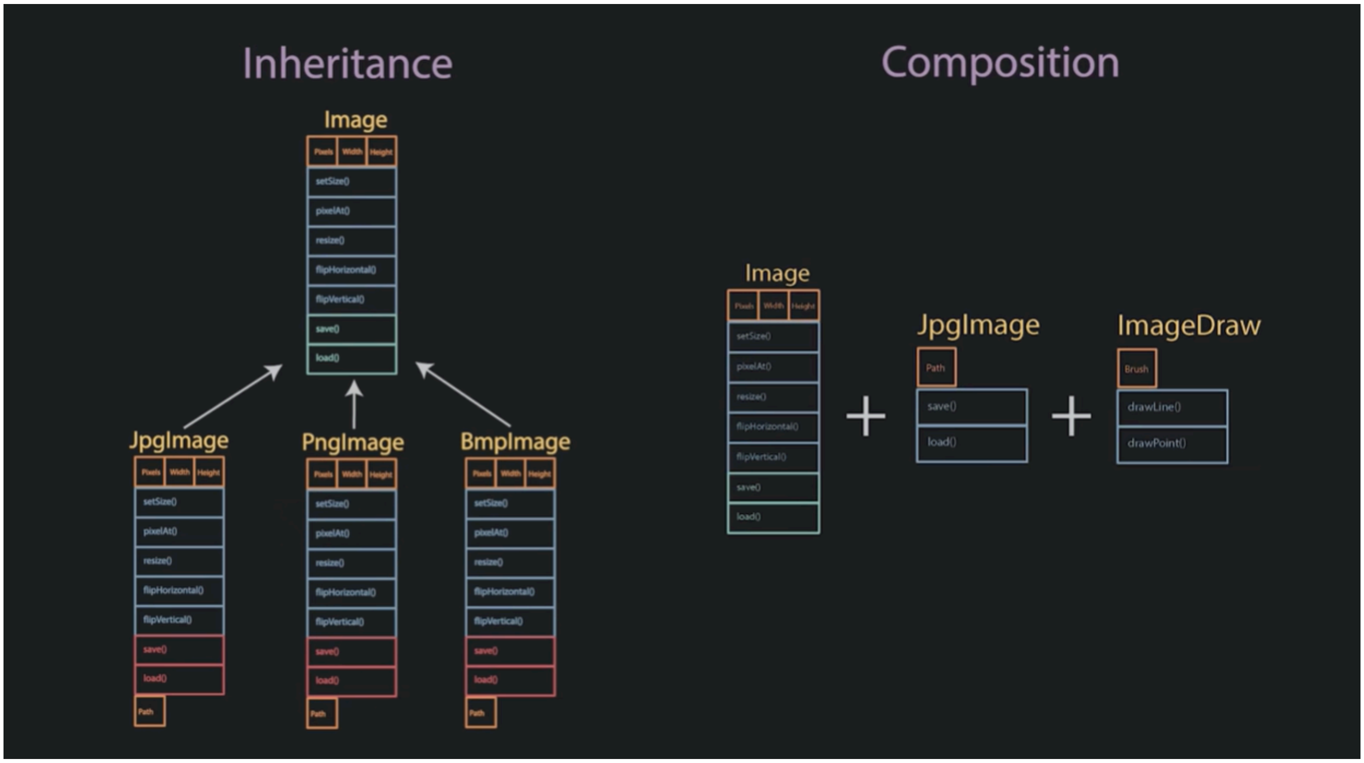
Now `Image` represents an image, and these other classes represent a specific file format.

Now if our new drawing requirement comes in. We create a new `ImageDraw` class that takes an image to draw to.

```
class ImageDraw {
    private Image image;
    private Pencil brush;

    public ImageDraw(Image image){
        this.image = image;
        this.brush = new Pencil();
    }

    public void drawLine(int startX, int startY, int endX, int endY) { ... }

    public void drawPoint(int x, int y) {
        this.brush.drawPoint(x, y, image);
    }
}
```

We are no longer bundled to the file stuff. Because we haven't bundled all of our similar elements into a parent class, we don't need to alter any of the other classes to draw an image.

> **Through composition**, we can combine classes together for any particular use case.



Here we're loading a `JpegImage`, loading it, and drawing to it.

```
{
    Image image = new Image

    JpgImage jpeg = new JpgImage("image.jpg", new JpegOptions());
    jpeg.load(image)

    ImageDraw draw = new ImageDraw(image);
    draw.drawLine(10, 10, 50, 50);
    draw.drawLine(50, 50, 10, 100);

    jpeg.save(image);
}
```

Inheritance vs Composition

Inheritance combines two capabilities, ***reusability*** through extending classes, and ***abstraction*** through parent classes.

Creating abstractions allows a piece of **code to use another without knowing exactly what code it's using.**

- Inheritance does this by allowing a *consumer* to think that it's given a class, but it's given a subclass instead.

Composition combines ***reusability*** through using classes (in other classes), and ***abstraction*** through **interfaces**.

- **Interfaces** simply describes the *contract* of what an object can do. They define only the critical parts of the contract.
    - They're easily added to existing classes.

Parent classes share everything by default, making them difficult to change. But **Interfaces are minimal**, they only include the **critical** parts of the contract.

```
interface ImageFile { // represents the operations an ImageFile can do.
    void save(Image image);
    void load(Image image);
}

class JpgImage : ImageFile { // we simply implement the interface!
    private string path;
    private JpegOptions options;

    public JpgImage(string path, JpegOptions options) {
        this.path = path;
        this.options = options;
    }

    public override void save(Image image) { // we now pass in an Image
        JpegEncoder encoder = new JpegEncoder(options);
        using (FileStream stream = File.OpenWrite(path)) {
            encoder.Encode<RGB>(this, stream);
        }
    }

    public override void load(Image image) { // we now pass in an Image
        JpegDecoder decoder = new JpegDecoder();
        using (FileStream stream = File.OpenRead(path)) {
            Pixel[,] pixels = decoder.Decode<RGB>(options, stream);
            this.replacePixels(pixels);
            this.setSize(pixels.GetLength(1), pixels.GetLength(0));
        }
    }
}
```

The Cons of Composition

1. **Code Repetition** in interface implementations to initialize internal types.
2. **Wrapper Methods** to expose information from internal types

Example of a Wrapper Method: These are when you return a call to an inner type.

```
void getName(){
    return user.getName()
}
```

The Pros of Composition

1. **Reduces coupling** to reused code.
2. **Adaptability** as new requirements come in

Overall composition grants more flexibility!!

When to use Inheritance

If you have an existing codebase that uses inheritance and you just need to make a small change, but changing a codebase might be costly so sticking with inheritance may be your most fruitful option.

Tips for Inheritance

Design the parent class to be inherited.

1. **Avoid protected** member variables with direct access
2. **Create** a protected API for children classes to use
3. **Mark** all other methods as private, final or sealed.
    - This presents bugs when changing your parent classes because you don't know what your child classes have done.

Venkat's opinion on Composition

Contrary to CodeAesthetic's video, Venkat is not a fan of composition at all. He suggests to use it very sparingly. A composition means a particular object owns a particular object but their

lifetime is tied together.

- When you destroy the owning object, you'll destroy the owned object.

```
class Brain {};
class Heart {};

class Person {
    Brain brain;
}
```

In this particular case, since the creation of `Brain` is attached to the creation of `Person`, that instance of person can never have a "`BetterBrain`", it's stuck with that one instance of `Brain`.

Composition doesn't give you good **extensibility**.

- The lifetime of objects are tied together.
- You cannot replace or change the object.

Aggregation is more **extensible** than composition.