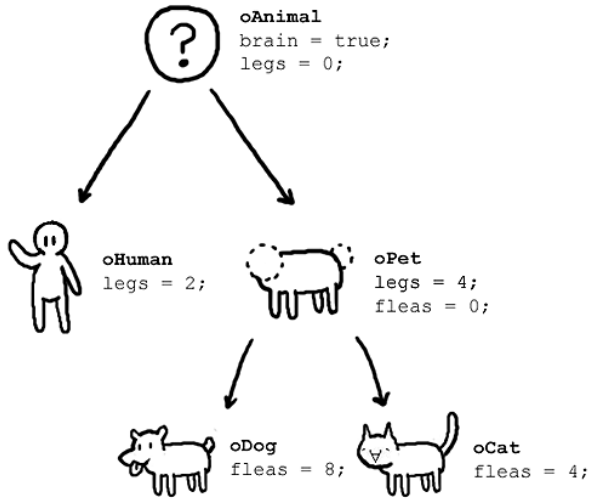
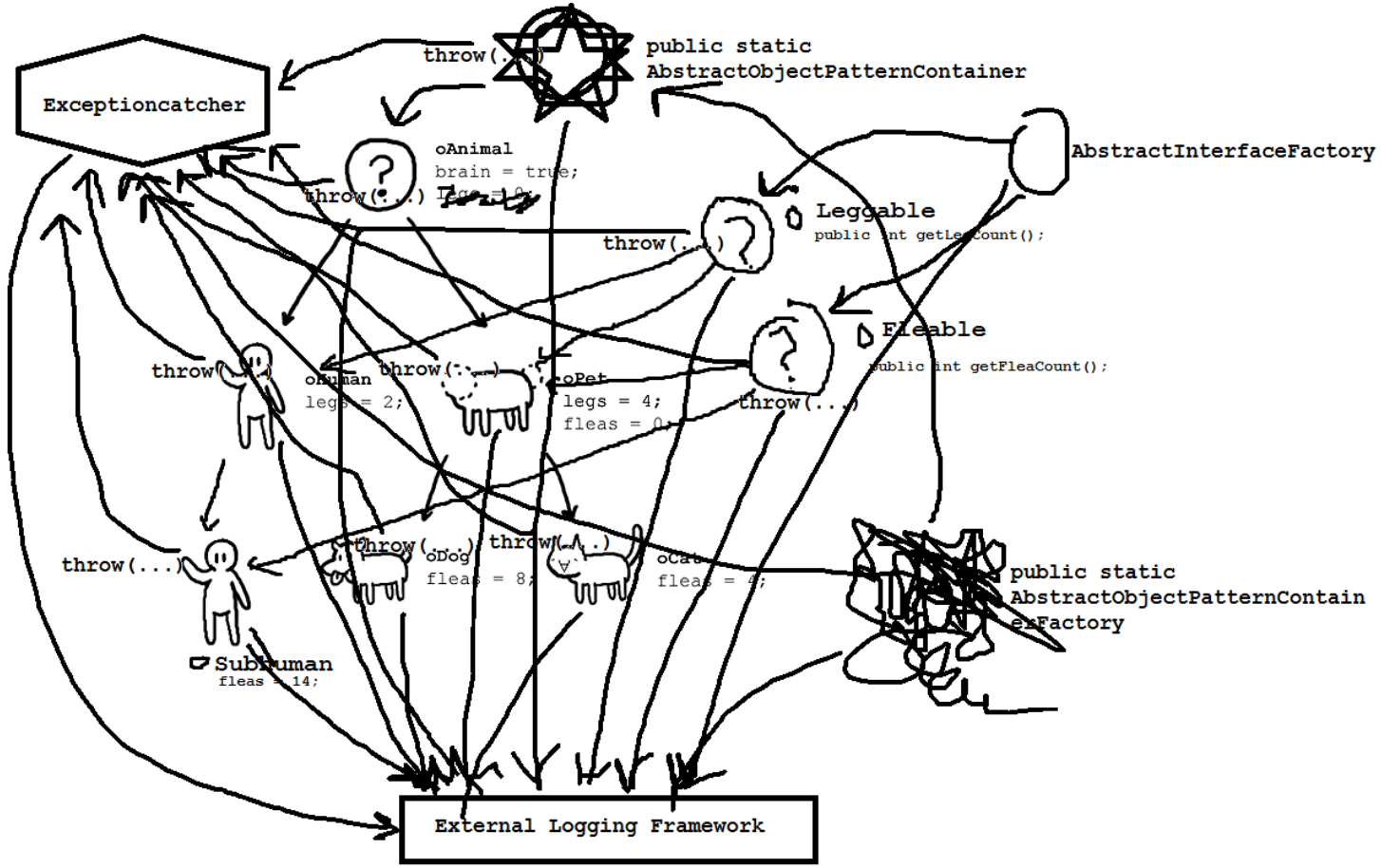
Lab 4: Inheritance & Polymorphism

What OOP users claim



What actually happens



Quick Recap from Lab 3

Understanding OOP by comparing it with other programming paradigms

- OOP is done at a **higher** level of abstraction (objects)
  - Objects communicate with each other using messages sent via methods (≈ verbs)
  - Methods usually modify attributes (≈ nouns)
  - Thus, it's great for situations that require mutable states (i.e. attributes)
- Attributes and Methods are **more closely knitted** vs procedural programming
  - Could have implemented Lab 3 tasks via procedural programming too
  - However, it becomes messy and unmaintainable for large scale programs
- Still based on imperative programming (you tell it **how** to do what you want to do)

OOP Principles

- **A**bstraction
  - Essential characteristics of object that distinguish it from others
- **P**olymorphism
  - Same message sent to the object, but each can do different stuff
- **I**nheritance
  - Child learns from parent (class), but child can do their own things too
- **E**ncapsulation
  - Hiding impementation and protecting data: like how a capsule protects what's inside

OOP Principles in other paradigms

The 4 OOP principles aren't unique to OOP

- Abstraction and Encapsulation can be done without objects
- Inheritance can be simulated in Functional Programming via closures
- Functions can be overloaded to achieve polymorphism

OOP Principles in other paradigms

The 4 OOP principles aren't unique to OOP

- Abstraction and Encapsulation can be done without objects

Abstraction is a very generic concept that is more than just the implementation you see in Java (i.e. interfaces and abstract classes is only one way to define the level of abstraction to work at)

Encapsulation is actually pretty omnipresent:

- `#include <stdio.h>` in C, name mangling in Python
- Whenever you interact with an API, many things (data and other functions) are hidden from view
- Java and C++ allows you to implement encapsulation more explicitly via access modifiers

OOP Principles in other paradigms

The 4 OOP principles aren't unique to OOP

- Inheritance can be simulated in Functional Programming via closures

1. Refer to this for a good 3 minutes intro of functional programming, with code examples
2. Then read this to understand what a closure is.
3. Finally, read this to see how closures are related to inheritance.

Essentially, given a function A that returns function B, **closure** refers to the local variables defined / used in function B that persists although we have already exited from function A.

With closures, we can 'inherit' prior states / attributes and functions that have been defined and used in another function.

OOP Principles in other paradigms

The 4 OOP principles aren't unique to OOP

- Functions can be overloaded to achieve polymorphism

Polymorphism ≈ same message sent (e.g. move()), each do different stuff (e.g. fly, swim, walk)

This is achieved in Java via **method overriding**: subclass redefines methods defined in superclass.
But it isn't the only way to do it.

In method overloading (e.g Python), you can also create 1 function that will do different things.
By defining the method with default keyword arguments, you can call functions with varying number of arguments and do different things.

Value of OOP and when to use it

- Strengths

  If designed well, classes are easily reusable & extensible (very big if!)
  > faster and cheaper development ; innate modularity allows teamwork

- Weaknesses

  Not very memory efficient, more lines of code, not easy to do well IRL
  > poorly designed classes lead to spaghetti code

- Use it when objects are the most natural representation

  'OOP works just as well for representing abstract concepts, such as "UserAccount" or "PendingSale", as for visible interface elements like "Window" and "Button"'

- Hard to have complete foresight about how all the objects interact

  "A good programmer will use strategies from both paradigms together in the battle against complexity."

Lab 4 Deliverables

Submit 1 .zip file to NTULearn (under 'Assignments')

- Your zip file should contain **17 .java files**
  - Numbers.java, Sorting.java, Strings.java, SalesPerson.java, WeeklySales.java
  - Shape.java
  - Shape2DApp.java, Circle.java, Triangle.java, Sqaure.java, Rectangle.java
  - Shape3DApp.java, Sphere.java, Pyramid.java, Cuboid.java, Cone.java, Cylinder.java

- If you're using Eclipse and your first line contains `package ...`, **remove** it before submitting
- No need to submit the .class file
- Follow the filename convention specified on NTULearn

References and Additional Readings

This slide deck was created using deck.js and exported via decktape.
It's no longer maintained, but it's still usable and easy to pick up via examples.
(all you need is HTML - with CSS and JS, you can make even better slides - it supports Markdown too)

You'll see an example using **reveal.js** for Lab 5 :)

A large part of the Lab 3 and Lab 4 notes are built on top of issues mentioned here - it is worth a read.

- Slide 1 Image
- More discussions about abstraction
- Inheritance can lead to exploding class hierarchy can be addressed by the decorator pattern.
  Do read up on design patterns - you'll cover it in CZ2006 and CZ3003 too.
- You wanted a banana but you got a gorilla holding the banana
  (having the right amount of encapsulation reduces this issue, but other paradigms are cleaner)