Object Orientation

#MAKE PROGRAMMING GREAT AGAIN

An Intro to OO

THE WAY YOU'VE BEEN TAUGHT

The Software Crisis

- So back to the 1970s/80s
- Programming was slow, and cost a lot.
- New projects required exponentially more time and cost a *lot*.
- Huge teams were needed, software quality was terrible.
- It was joked that if normal engineering was done like software engineering, nothing would stay upright.

And I saw, and behold… *Spaghetti*

- Spaghetti coding was still an actual paradigm!
- Because "programming" was still a new profession, had bad review methodology.
- Made large software development actually impossible.
- Problem made worse by programmers not being willing to make changes
- Wouldn't accept languages that improve programming practices!
  - This is why nobody uses Lisp these days.

Viva La Structured Revolution!

- Software teams on larger projects noted that code reuse via structured programming was a good way to reduce programming time.
- However, this wasn't well addressed (think of Pascal!)
- Speed was usually a big problem in structured programs.
- And low level languages were attractive cause they were really fast!

Alternatives

- Modular Programming
  - ADA was the king of this, however ADA was designed for the US military, and so unless you were working on low level stuff it was still not flexible enough. (UI's are not needed on a missile!)
- Fourth Gen Systems
  - Think SQL and databasing. These are great, but suck at doing anything too fast.

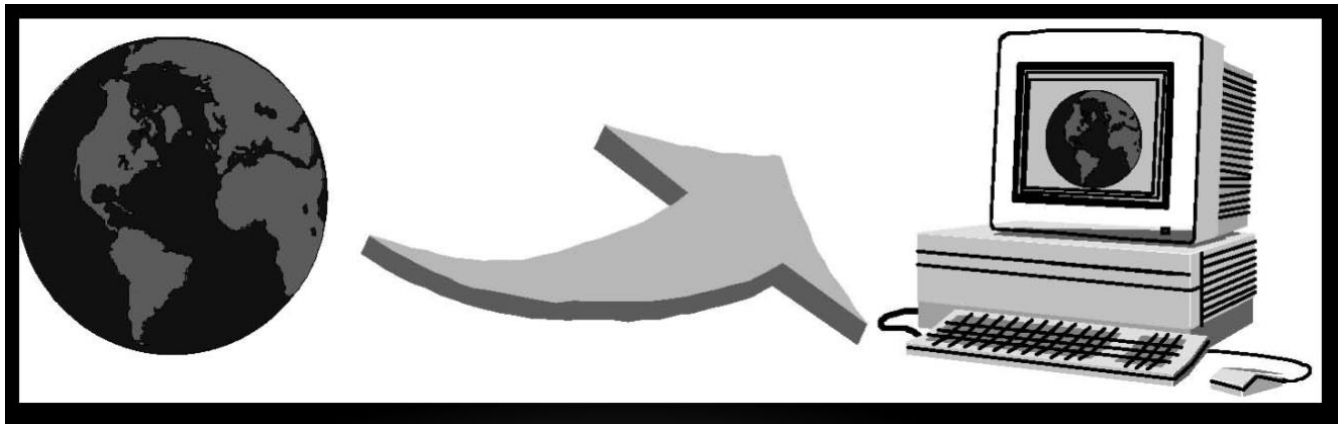
And so, someone invented OO

- Fixes the problems with low level languages
  - Fixes software reuse because ADT design is *forced*
  - Maintains independence between components by encouraging a detached, interface heavy design.
- OO is better suited to heavy computing than DBMS
  - OO provides a pointer safe environment (usually) and allows for high code reuse
  - Defines an "environment" rather than just a "language".
  - We need to know how these differ, it's important for studying OO.

OO Basics

- Software components can be modified for reuse in 5 ways in OO
  - Extend a components fields and functions.
  - Restrict a components fields and functions.
  - Redefine a function to do new things
  - Combine two different components to make a new one
  - Extend the type of data that a component function can work with.

The Basic Idea



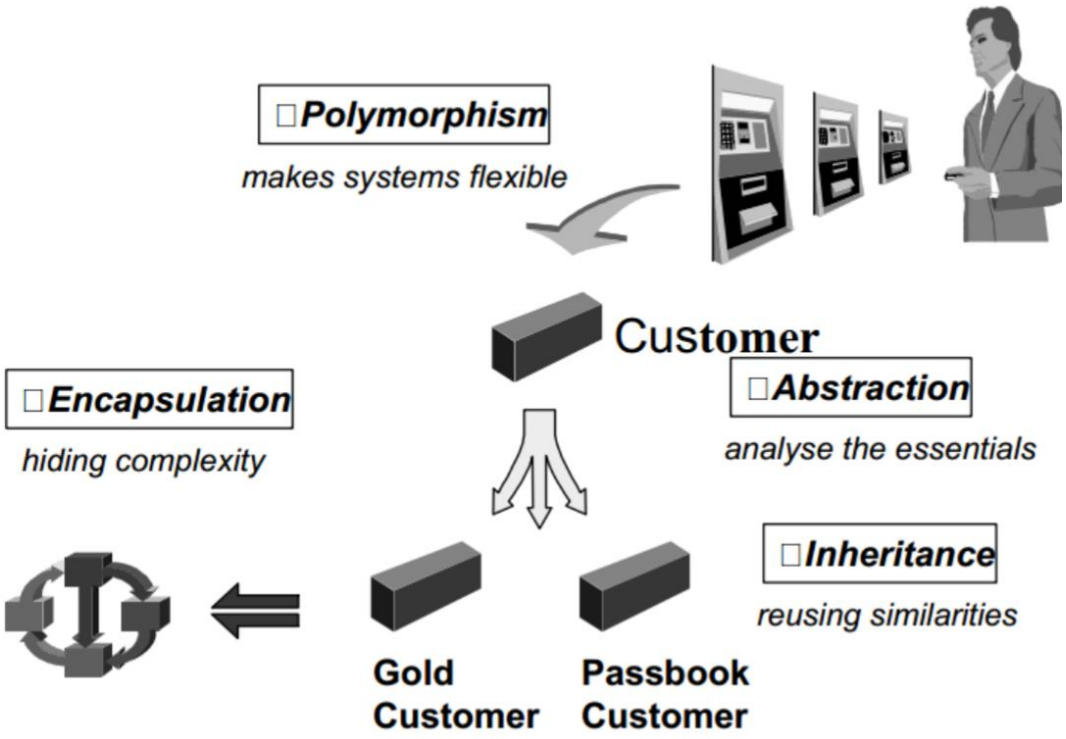
The Basic Idea

The Basic Idea

Seven Requirements of OO

1. Systems are modularised on the basis of their data structures
2. Objects should be described as implementations of abstract data types
3. Unused objects should be automatically de-allocated by the underlying environment
4. Every non-simple type is a module, and every module is a type
5. A class may be defined as either an extension of, or restriction of another.
6. Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realisations in different classes
7. It should be possible to declare a class as heir to more than one class, and more than once to the same class.

Major OO Concepts



Classes

- Are representations of real world things
  - Int class representing the number 3
- Has behaviors
  - Add: “Tell 3 to add 5 to itself”
  - Multiply: “Tell 8 to multiply itself by -2”
  - These are called “methods” or “features”
- These are all you need to model the world.

Class Abstraction

- OO requires you to class things by similar properties or similar behaviors
  - You don't make a "Stefan" class, you make a "person" class and make a "Stefan" instance of it.
- All objects of the same type share behaviour and representation

Classes V Objects

- Classes
- Generalisation of a set of objects
- Have properties
  - People have heights
- Have behaviors
  - People can walk
- Can create and destroy processes
  - People are born and die.

- Objects
- Are instances of classes
- Objects actually exist
  - You can't find a people.
- Classes have identity
  - Stefan is a person
- Has properties
  - Stefan has a height
- Has behaviors
  - Stefan can do lectures
- Created and destroyed

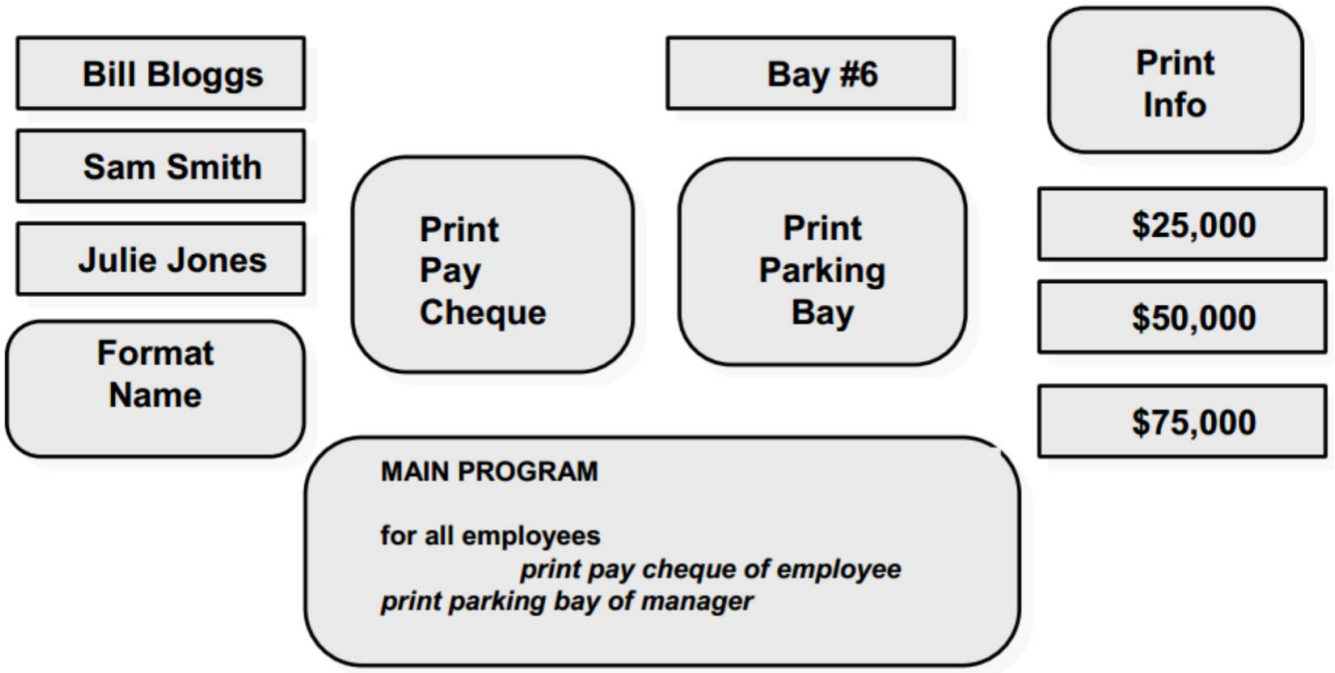
Objects Do Things

- Classes specify what objects *can* do.
- Objects do them
- Ie, tell car "GR8MATE" to drive to Uni.
  - Message is given to car
  - Message is "drive to"
  - Detail is "Uni"
- This "drive to" is specified in the car Class.
- Clearly takes in some kind of place object too as detail!

Paradigm Differences

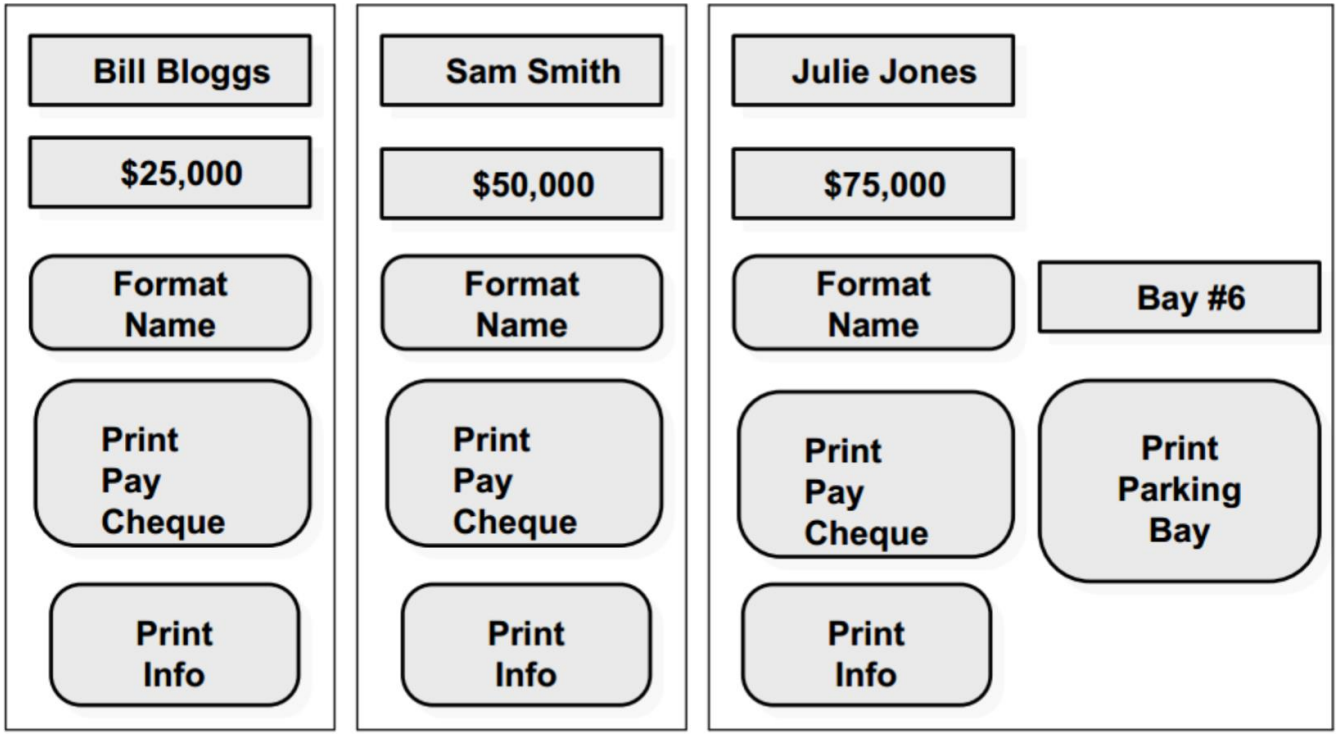
- Procedural:
  - Program = Data + Functions
  - Focuses on algorithms and separate data structures
  - Questions are:
    - What does this system do?
    - How do I write code for it?
- OO:
  - Program = Set of Objects
  - Focuses on…. Objects
  - Questions are:
    - What objects exist in this system?
    - How do these objects interact?

Traditional Code Example

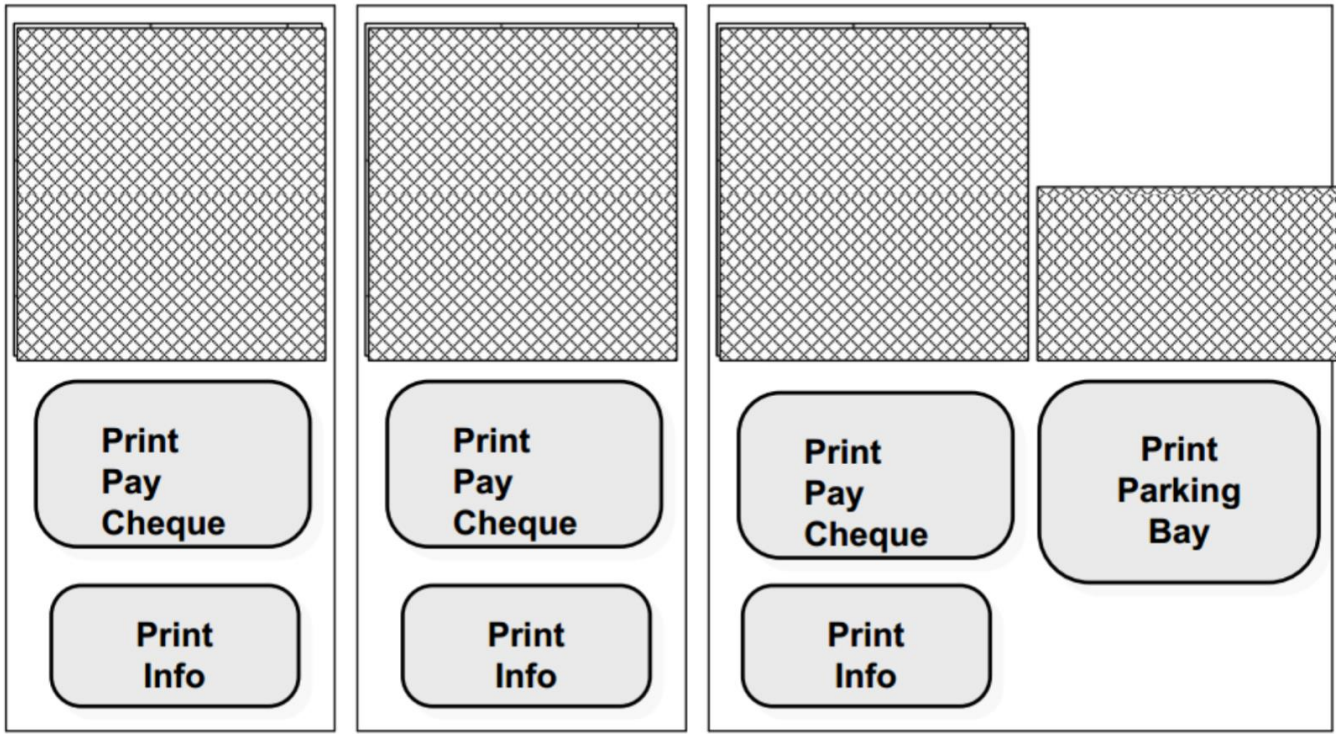


First, encapsulate into Objects

- Define things that are common as objects.
- In this case, employees.
  - Julie has a salary and a parking bay, can tell you her name and salary, can get a pay check, and can tell you what her parking bay is
  - Sam has a salary and no parking bay, and can tell you his name and salary, and can get a pay check
- Write a “Employee” class, specialise with a “manager” class


Bill Bloggs
$25,000
Format Name
Print Pay Cheque
Print Info
Sam Smith
$50,000
Format Name
Print Pay Cheque
Print Info
Julie Jones
$75,000
Format Name
Print Pay Cheque
Print Info
Bay #6
Print Parking Bay

Information Hiding



Encapsulation vs Info Hiding

- If you are hiding information, you don't neccisarily need encapsulation!
  - You could just be hiding functions, think private functions in C
- Encapsulation doesn't necessitate information hiding!
  - Public everything doesn't hide anything!
- So be aware that you can't infer one from the other!

OO Program Code

MAIN PROGRAM
for all employees
tell employee to *print pay cheque*

for all employees
tell employee to *print info*

tell *manager* to *print parking bay*

OO Programs

- Consist mostly of objects sending messages based on attributes.
- Messages are interactions where an object calls another object's methods.
  - May be it's own methods, but whatever
- Attributes may be properties or ephemeral
  - BMI may not be stored, it may be calculated when asked for.
- All together, objects provide "services" in terms of their methods and attributes that are public to other objects.

Wait, public and private?

- Public:
  - Service names (interfaces)
  - Properties
  - Operations (methods)
- Private:
  - Attributes
  - Behaviors

Views of a Class

- These provide different views of a class depending on whether or not you are an object of that class
  - Sometimes you can go further when you're a class child
- Messages cause objects to act
  - Generally consist of the object name, the service and if required optional arguments.
    - GR8MATE.driveto(Uni);
- What's different between this and just writing a function?

Modularisation vs OO

- Modularisation is cool, it reduces a programs size and complexity via structure and reuse.
- OO goes further by collecting together everything within a program that might alter the data into isolated objects.
- The effects these have on the data are therefore predictable, and entirely reusable.

Classification

- Process of creating classes for real world objects.
- You need to look at what is common to a group of individual things that will be interacting in your system
- Examples:
  - Real numbers
  - Employees
  - Cars

Specialisation

- Specialisation classes (named children, cause of inheritance) have everything that their generalisation (parent) has.
- Like a big inverted pyramid, at the top is most general, at the bottom is least general.
- Any changes you make to a child class reduces its generality.
  - This can include adding things, but they invariably are for specific purposes.
- Any child class is a kind of their parent, and can be used as such

ADT's

- Classification creates Abstract Data Types by definition.
- From Ada, it's a data structure and everything that you need to make it run
- ADT's *are* classes!
  - In OO. Only in OO environments.

OO Environments

- Consists of a “System
  - A set of objects
- What objects comprise your system?
- How can you model the system using the objects and their behaviours?
- This removes initial thought about what algorithms you’ll need, you only care about how these things will interact

OO relationships

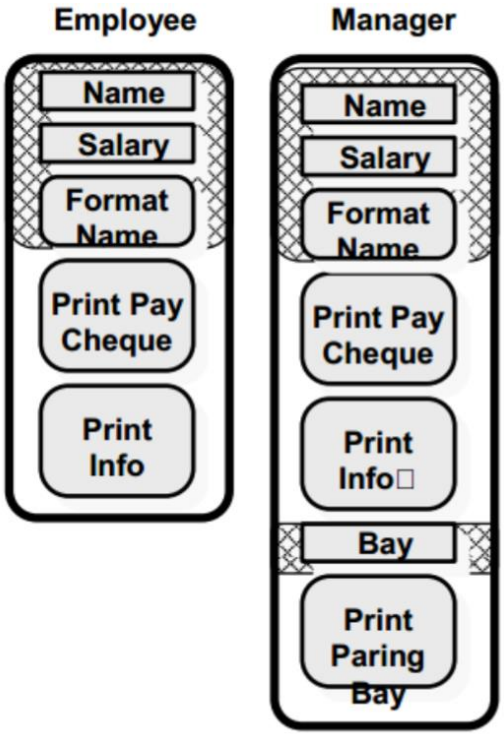
- There are four:

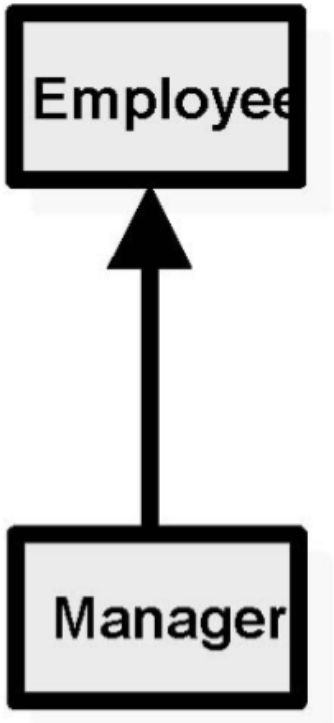
- Inheritance
- Aggregation
- Association
- Friendship

Friendship? What are these?

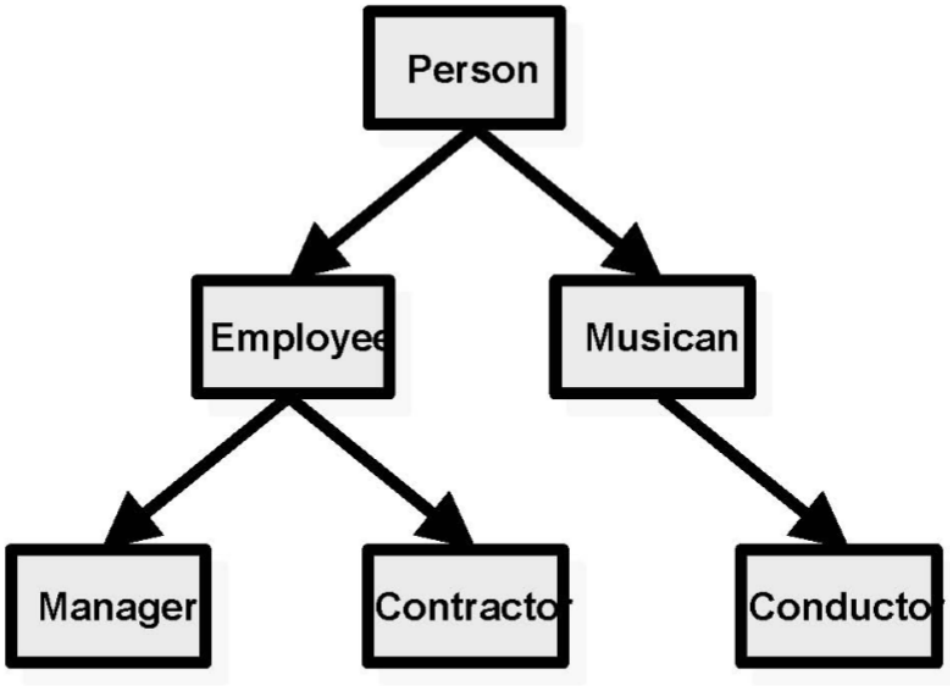
- Inheritance:
  - Specialisation of a class. May have new properties, services, etc.
- Aggregation:
  - Is a part of: When an object is embedded within another class. They are different classes, but can't exist without eachother. Think Engine and Car.
- Association:
  - Uses: when the objects make use of one another. They aren't a part of eachother.
- Friendship:
  - When one class needs to know about another class. Usually when it needs access to it's components.

Inheritance

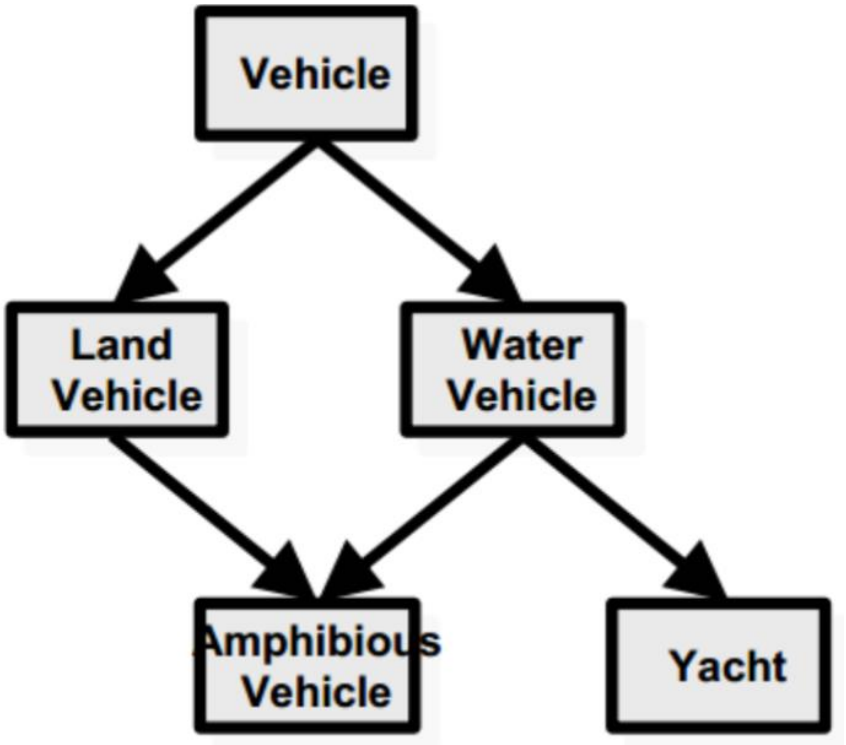




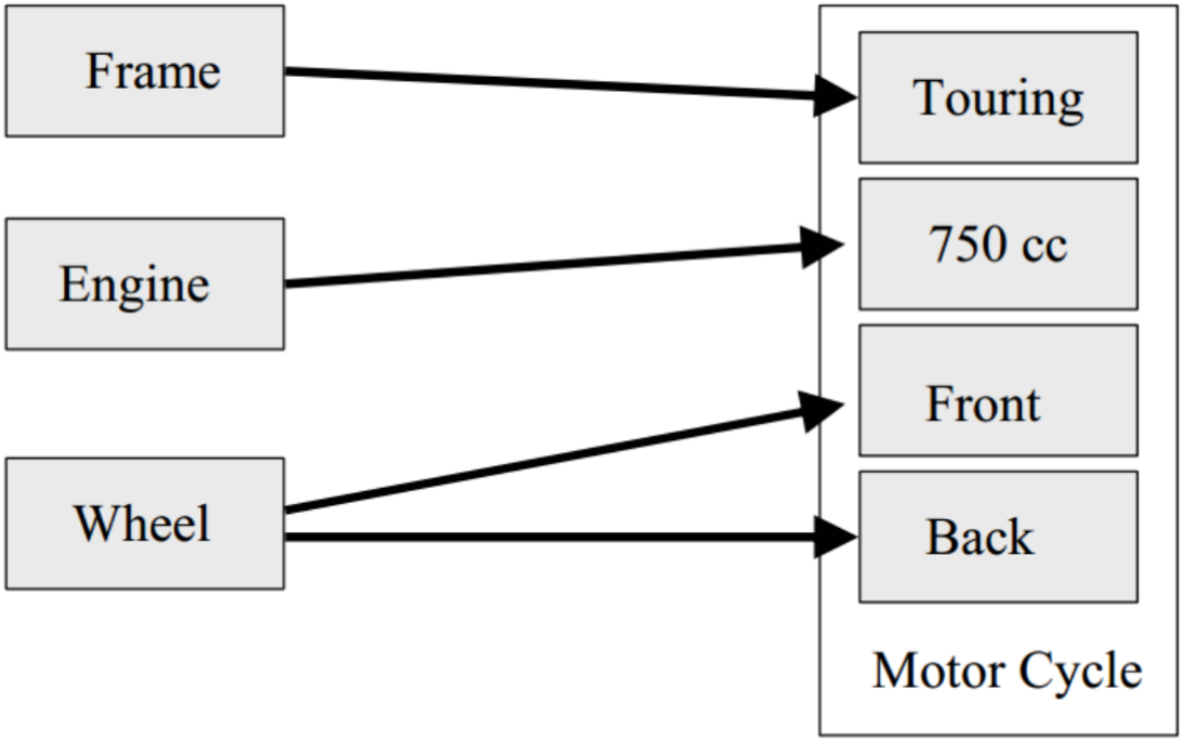
Single Inheritance



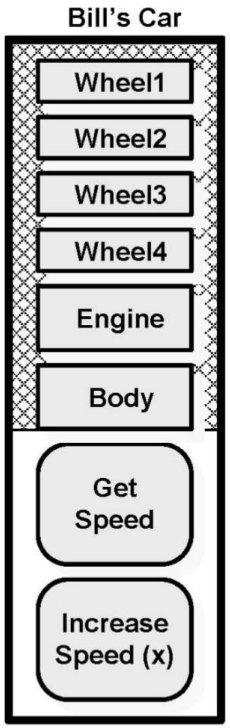
Multiple Inheritance



Repeated Inheritance



Aggregation



Polymorphism

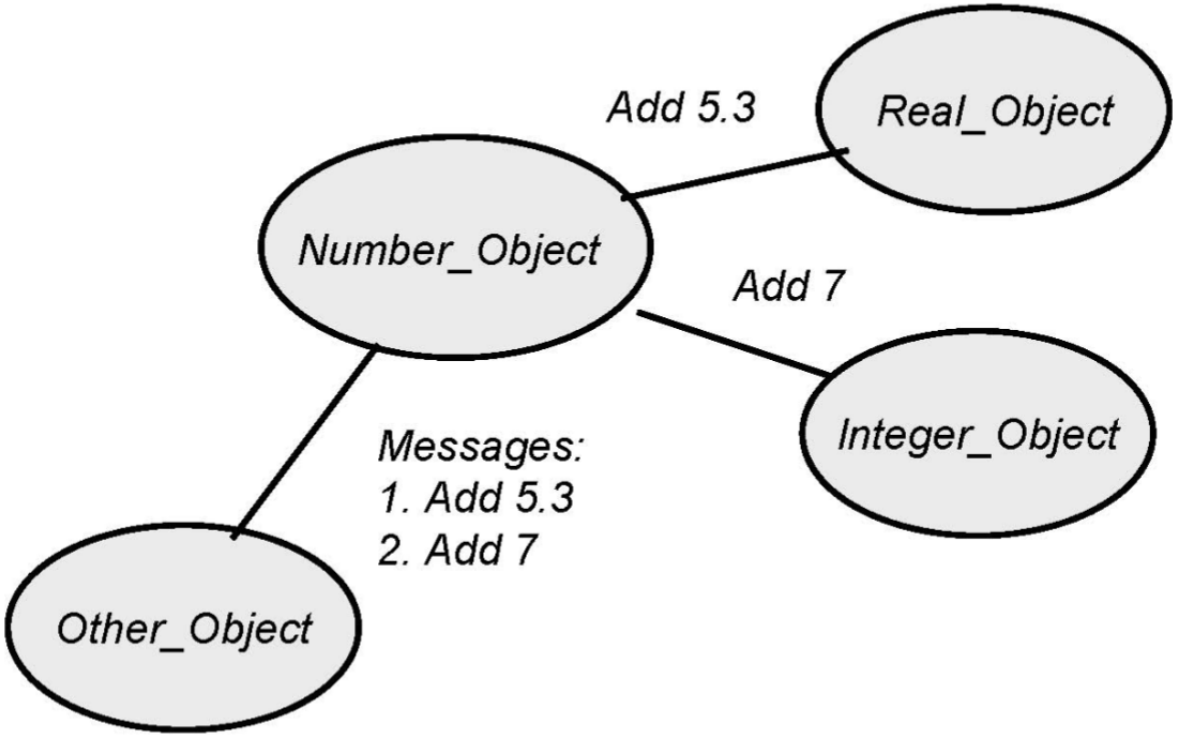
- Means many forms.
- Same message, different behaviour
- Also called operator overloading
  - You can add two reals and two ints
- Two kinds:
  - Static (compile time) Binding
  - Dynamic (run time) Binding
- Polymorphism is *usually* implemented via inheritance

Static binding

- Can tell objects to respond to the same message differently
- Means I can send an int or a real to "add" and it won't throw an exception

```
...
tell known object alpha to do service a
        using x
tell known object alpha to do service a
        using y
...
```

Static Example



Dynamic Binding

- Send messages to an object without knowing what it is
- No reflection or switch/case coding needed!

```
...
for all unknown objects in the bag
        tell unknown object to perform service
...
```

Dynamic Binding Example

```
class Shape {
        virtual int Draw(){ cout << Shape << endl;
}
class CIRCLE {
        virtual int Draw() { cout << Circle << endl;
}
Circle* c, *c->calls CIRCLE's draw
Shape* s,
        if *s is a Shape *s->draw calls Circle's draw
        (if s in fact is pointing to a Circle)
```

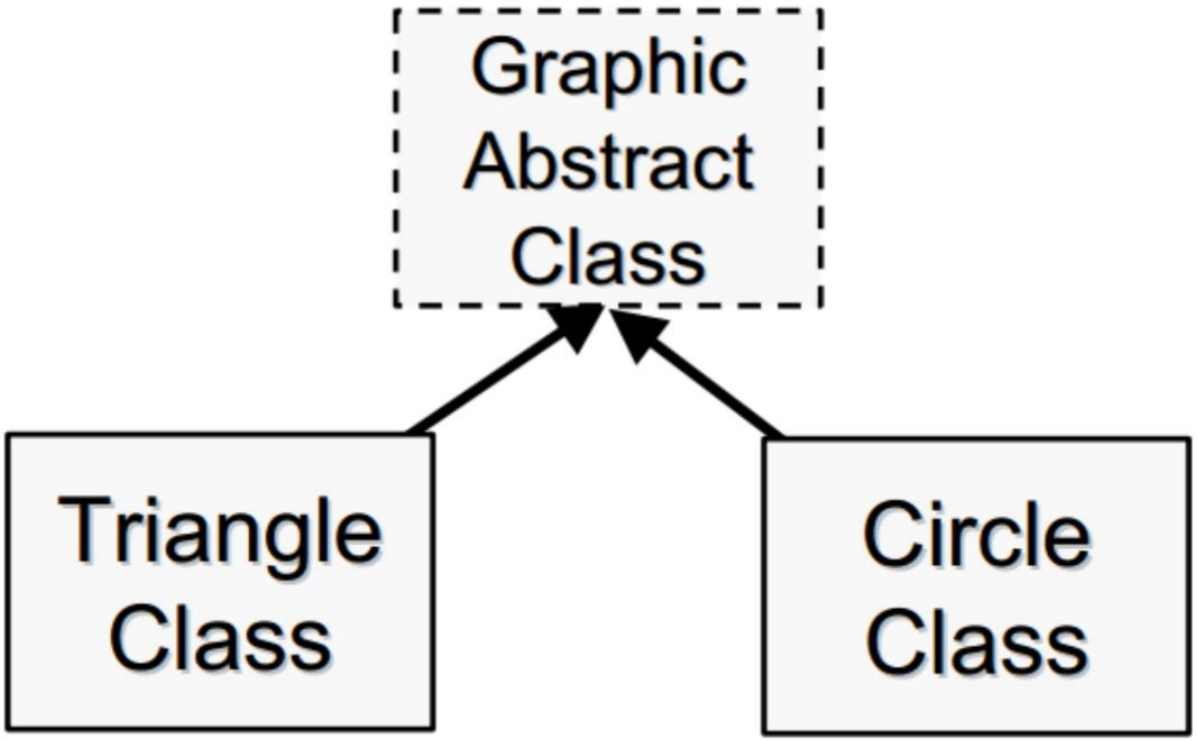
Why Dynamic?

- Can just add new classes, no problem
- Simply write new versions of old methods
- The decision of which method to perform is done at run time based on the class of the object
  - No recompilation!
  - Just works like magic.
  - Consequentially, this is how windows works.

Abstract or Deferred Classes

- Classes which cannot or are not instantiated because they don't actually have a full specification
- Usually are some abstract concept
  - Useful for redundancy in specification of classes
  - Provide stub classes from which to derive actual classes
  - Can relate classes that have no physical or actively modelable relationship (vehicles, for example)

Abstract/Deferred Example



Watch out though, OO != OB

- Object based is not the same as object oriented
  - OO does not emphasise process over data
  - Correct design is crucial
- OO is not necessary for things like:
  - GUI's
  - Microsoft OLE