Objects & Traits

- Most of the programming languages we use to today are inheritance based OO languages
- The main distinguishing feature between them is whether they support single or multiple inheritance.
  - C++ and Python support multiple inheritance
  - Java support single inheritance
- There are three main problems with inheritance based OO languages.

Problem #1

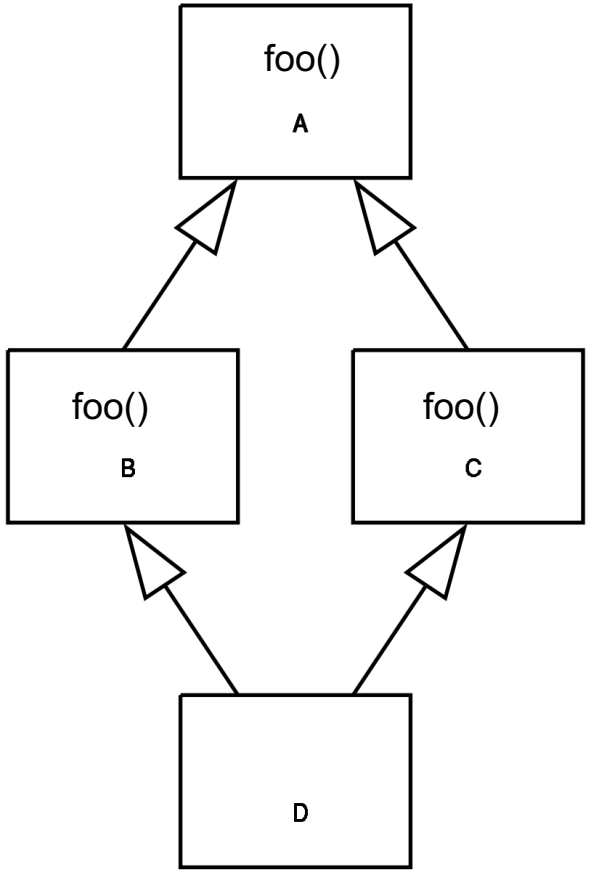
- *Bloated method inheritance* – that is, each child in an inheritance hierarchy will inherit ALL of the methods of its ancestors.
- This is true for both single and multiple inheritance.

Problem #2

- The *diamond problem* – sometimes referred to as the ‘deadly diamond of death’
- This occurs in languages with multiple inheritance

The Diamond Problem

- Briefly:
  - An ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
  - If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?
  - That is: D.foo() – which foo() should be called?
- This gets really problematic in deep inheritance structures.



The Diamond Problem

- Different languages deal with the diamond problem in different ways
  - C++ uses a fully qualified syntax
  - Python uses a class hierarchy linearization algorithm (C3 linearization or MRO) to resolve ambiguities

CSC301-git — python3 — 117×21

```
(base) MacBook:CSC301-git lutz$ python3
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(B,C):
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>>
```

MRO: Method Resolution Order

Problem #3

- A third problem that frequently arises in inheritance based OO languages is *rigid class structures*
  - This usually manifests itself in class hierarchies that are difficult to evolve in face of changing software requirements

Rust

- Rust solves all three of these problems by getting rid of inheritance all together and introducing *traits*
- Technically traits are ‘mixins’ - a class that contains methods for use by other classes *without having to be the parent class of those other classes,* that is, without using inheritance.*
- Traits give you:
  - Better evolvability of your object structure
  - Better control of what your object interfaces actually look like – no method bloat.

*https://en.wikipedia.org/wiki/Mixin

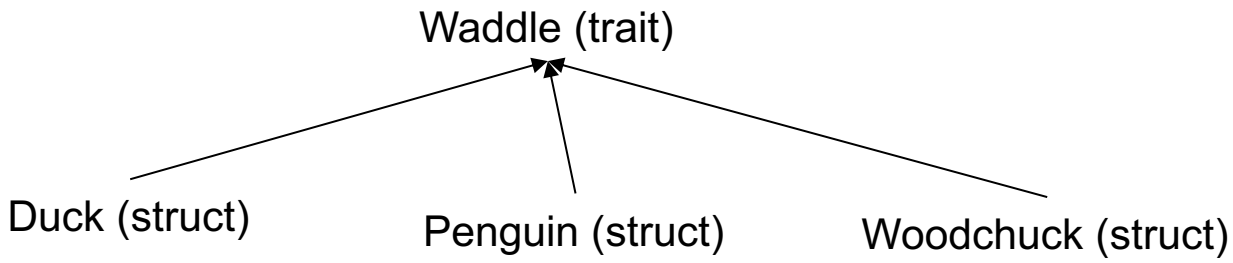
Traits

```
1  // Define some traits we want our objects to have
2  trait Quacks {
3    fn quacks (&self); // abstract function - no body
4  }
5
6  trait Waddles {
7      fn waddles (&self); // abstract function - no body
8  }
9
10 // Define our object and implement the traits
11 struct Duck { name : String }
12
13 impl Quacks for Duck {
14     fn quacks(&self) { println!("{} quacks!",self.name)}
15 }
16
17 impl Waddles for Duck {
18     fn waddles(&self) { println!("{} waddles!",self.name)}
19 }
20
21 // Instantiate an object and test drive the traits.
22 fn main() {
23   let polly = Duck {name: "Polly".to_string()};
24
25   polly.quacks();
26   polly.waddles();
27 }
```

Screenshot

Polymorphic Programming

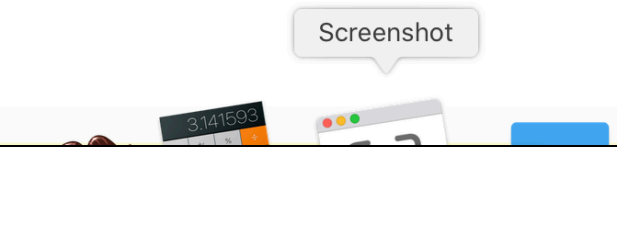
- Traits introduce types
- Implementing a trait for an object class means we are creating a subtype-supertype relationship
- Turns out that in Rust this is the only place where a subtype-supertype relationship exists
- Example:



Notation: A → B means A implements trait B.

Polymorphic Programming

```
1  // Define a trait we want our objects to have
2  trait Waddles {
3      fn waddles (&self);
4  }
5  // Define our objects and implement the traits
6  struct Duck { name : String }
7  impl Waddles for Duck {
8      fn waddles(&self) { println!("{} the duck waddles on land", self.name);}
9  }
10 struct Penguin { name : String }
11 impl Waddles for Penguin {
12     fn waddles(&self) { println!("{} the penguin waddles on ice", self.name);}
13 }
14 struct Woodchuck { name : String }
15 impl Waddles for Woodchuck {
16     fn waddles(&self) { println!("{} the woodchuck waddles low to the ground", self.name);}
17 }
18 // polymorphic programming with traits
19 fn main() {
20     let animals: [&Waddles;3] = [
21         &Duck {name: "Polly".to_string()},
22         &Penguin {name: "Schubert".to_string()},
23         &Woodchuck {name: "Wally".to_string()}
24     ];
25
26     for i in 0..animals.len() {
27         animals[i].waddles();
28     }
29   }
```



Assignments

- Assignment #3 – see BrightSpace