Week 6

Brendon Faleiro

Large Scale Programming Structures

Modules

- Separation of interface from implementations.
- The interface is a definition of what operations a certain object can perform.

- OOP languages allow you to define variables as private or protected.
- Ocaml has modules.

- Why would you want to hide implementations from the user?
  - Enforce what the user can do.
  - Get the flexibility to change the implementation without breaking the API

```
module type MODTYPE = sig
type ‘a modtype
            :
            :
(*Type definitions that are publicly visible and accessible.*)
            :
            :
end;

module ModName : MODTYPE = struct
type ‘a modtype = ……
            :
            :
(*actual implementations*)
            :
            :
            :
```

Moving on to JAVA.

Java Interfaces

- An “interface” declares what operations the API will support. It is a basic minimum set of operations that any “class” that aspires to be of the same type as the interface must implement.

- A class “implements” an interface. It must create definitions for each of the methods that the interface states it should.

- A “class” is a template for object creation. It defines what data must be created for each object and what methods are accessible to the object.

- Now, when I define a class I don’t know what my object will be named. But all methods in the class will be called by some sort of instantiation of that class. We use the “this” keyword in a class to refer to the current instantiation or object that we are working with.

Java Inheritance

- Java supports inheritance using the extends keyword.
- A class in Java can implement any number of interfaces, but I can only inherit from one class.
- Unlike C++, Java does not support multiple inheritance.
- Inheritance is used to **avoid code duplication**.

Subtype Polymorphism

- Any class that implements a certain interface or extends a particular class is said to be a subtype of that class.
- A subtype can do everything that a parent type can do and more.
- You can always pass a subtype wherever a parent type is wanted.
- However, the reverse is not true.
- The parent usually has lesser abilities than the subtype and hence cannot be passed instead of the subtype.

- Subtypes are substitutable for supertypes.
  - Instances of subtype won't surprise client by failing to meet guarantees made in supertype's specification.
  - Instances of subtype won't surprise client by having expectations not mentioned in supertype's specification.
  - If B <: A, anything provable about an A is provable about a B.
  - If instance of subtype is treated purely as supertype – only supertype methods and fields queried – then result should be consistent with an object of the supertype being manipulated.

Types vs Classes

- An object's **class** defines how the object is built. The class defines object's internal state and the implementation of its operations.

- In contrast, an object's **type** only refers to its interface -the set of requests to which it can respond.

- An object can have many types, and objects of different classes can have the same type.

- Substitution (subtype)
- B is a subtype of A iff an object of B can behave as an object of A in any context.

- Inheritance (subclass)
- Abstract out repeated code.
- Enables incremental changes to classes

- Every subclass is a Java subtype
- But not necessarily a true subtype. A true subtype is one where the subtype has stronger restrictions than the parent type.

Eg. Dog <: Animal

Animal a = new Dog();

Q. Which of these would work???

a) a.walk();

b) a.eat();

c) a.bark();

(a) and (b) work. Because “a” is an animal and any animal can walk or eat. But “a” could be any animal and it wont necessarily know how to bark.

- Eg. You have the classes Bird, Duck, Cuckoo. What should be the expected subtyping relationship between the above classes?

a) Bird <: Duck, Bird <: Cuckoo

b) Duck <: Bird, Cuckoo <: Bird

Ans: (b)

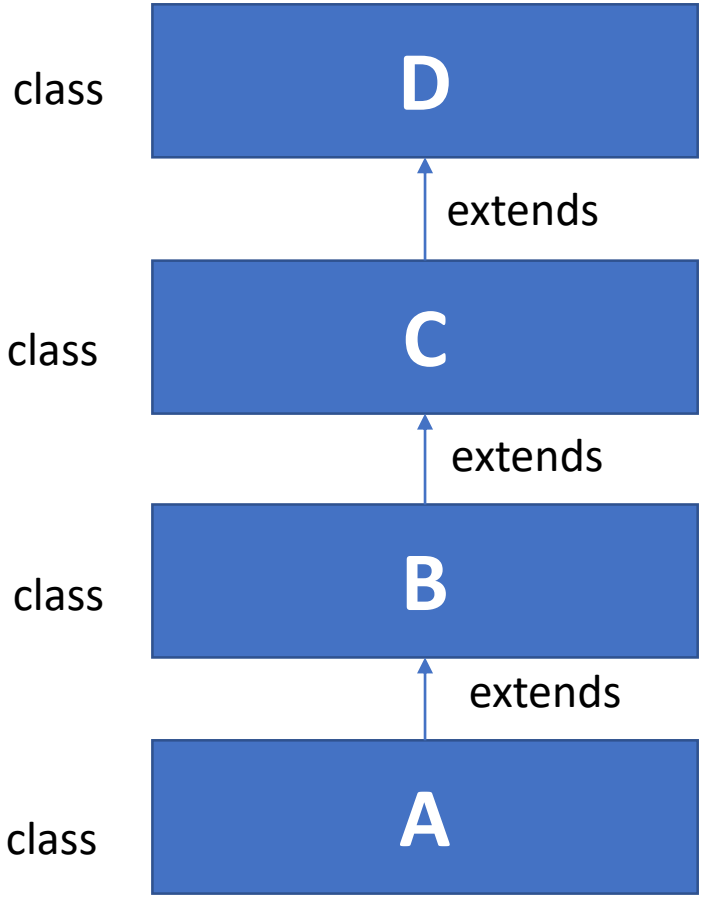
Whatever a bird can do, a duck or a cuckoo can do. But not vice versa! Not all birds can swim (but ducks can), nor can all birds sing (cuckoos can). So whenever I need a Bird, I can pass a duck or cuckoo. But I cant pass a Bird when I specifically need a cuckoo.

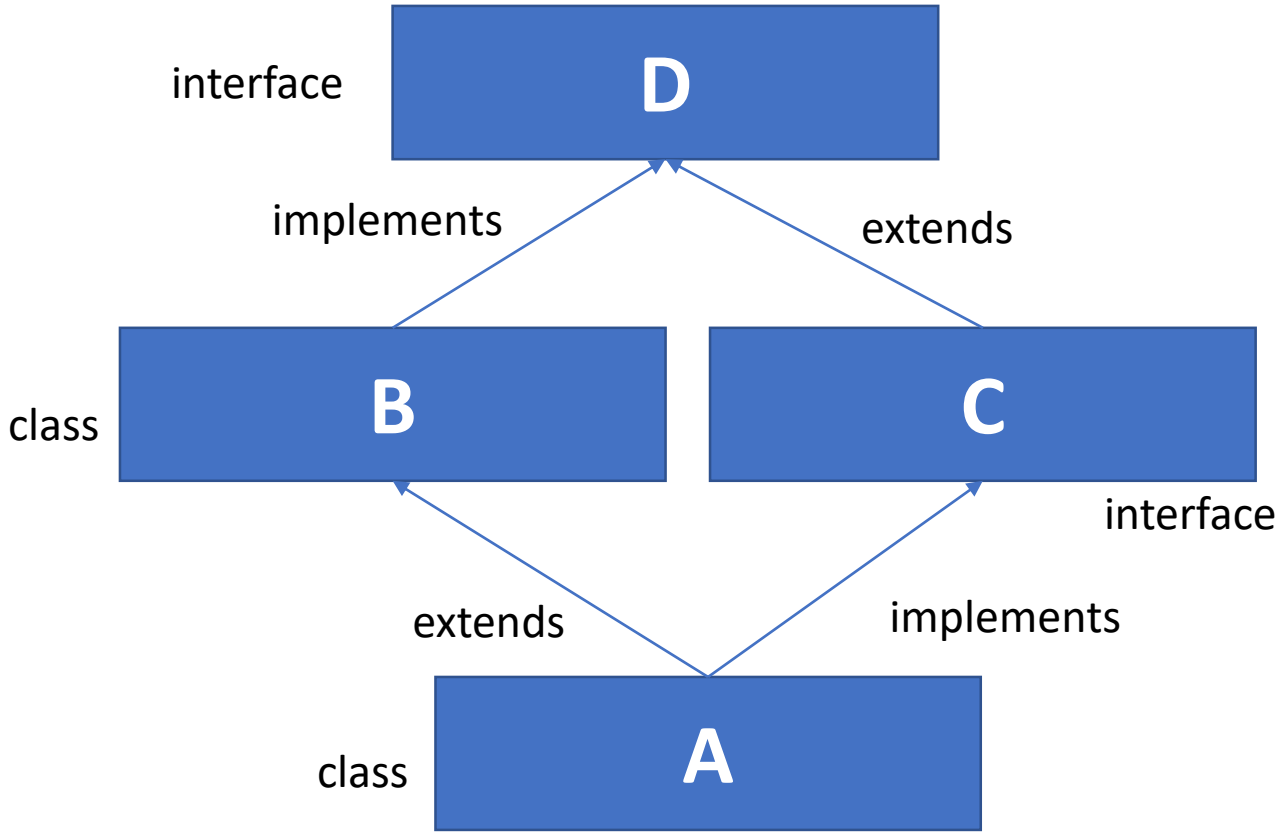
IMPORTANT!!

- Inheritance has nothing to do with subtyping.
- Subtyping polymorphism is orthogonal to inheritance.
- Subtyping is about interface compatibility.
- If S <: T, then I can pass S in place of T.
- Inheritance looks at implementations.
- If C inherits from D, then C has all the code that comes with D.

- Java is a special case:
  - interface C extends D -- C is a subtype of D
  - class C implements D -- C has the type of D
  - class C extends D -- C inherits D, C is a subtype of D.

- Give an example of four distinct Java types A, B, C, D such that A is a subtype of B, A is a subtype of C, B is a subtype of D, and C is a subtype of D. Or, if such an example is impossible, explain why not.





These are just two possibilities. There are several other implementations possible.

Q. Given that S <: T.

Is C<S> <: C<T>?

Ans. No.

Is C<S> was a subtype of C<T>, then I would be able to pass C<S> anywhere where C<T> was expected.

Now, suppose some other class X <: T, then I can add instances of type X to C<S> (by assuming it to be C<T>).

But X and S may be incompatible and cause serious issues.