Inheritance, subtyping and polymorphism

Perdita Stevens, University of Edinburgh

March 2010

Is a kind of

You're thinking about the objects in your zoo management system. You spot two tigers, one lion, one elephant, three penguins, ...

Where do you draw the class boundaries?

What distinctions does the system need to make?

Different contexts, different distinctions

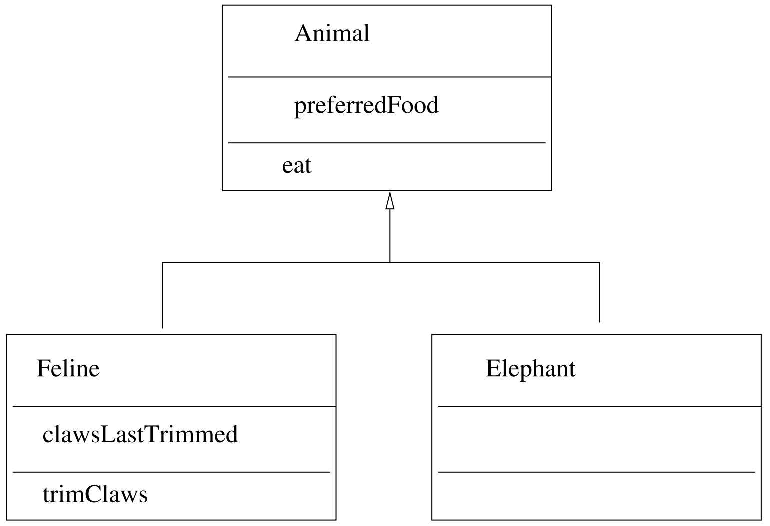
If lions are *always* treated the same way as tigers in the system, they can share a class (Felines?)

Things which the system treats *totally* differently can live in separate classes, *even if in the domain there's a relationship* (Judgement call! Keep it simple...)

What if it depends on the context?

E.g. penguins, felines and elephants all need to be fed, but they eat different things, and only felines need their claws trimmed.

Possible subtype hierarchy



What difference does it make?

Compare

A: Elephant and Feline unrelated classes, implementing eat separately

B: Elephant and Feline subclasses of Animal, which implements eat

The choice affects:

1. Clients of the classes that do not need to distinguish Elephants from Felines. In A, such clients will have to duplicate code.
2. The classes themselves. In A, Elephant and Feline *may* duplicate code.

1 is far more important and far more convincing as a motivation for the hierarchy.

Subtyping

Even if you don't implement all the real world relationships, be consistent with them.

Felines may have all the state and behaviour of Elephants, and then some, but a Feline is not an Elephant.

"Every x can be regarded as a y"

"An x is a kind of y"

"An x is a y" – commonest, but sometimes misleading

Consider Labrador, Dog, Breed...

Substitutability

Suppose T is a subtype of S ("every T can be regarded as an S")

Then, any code written to work with arbitrary Ss *can reasonably be expected to work* when given a T.

What's in the "can be regarded as" though? A T may understand all the same messages that every S is expected to understand, but remember, what it does is encapsulated.

And what's "work"?

(Later, we will discuss Liskov Substitutivity)

Polymorphism

Code is polymorphic if it works with "many shapes" of objects – with more than one type of object.

In popular class-based statically-typed OO languages the main example is that code written in terms of a class C will also just work with objects from any subclass of C.

IOW, the compiler/run time system ensure that the right code is executed and the client code doesn't need to know.

Example

```
takeCareOf(Animal a) {
   a.eat("universalfood");
}
```

Even though the client has never heard of Elephants or Felines, the right code will be invoked, depending on what a is (at runtime – this is "dynamic binding" – but possibly determined at compile time – "static typing for dynamic binding").

Specifically this is *subtype polymorphism* – we'll come across another kind later.

Inheritance

In our languages, subtyping is implemented using inheritance.

Suppose class S exists, with some state and behaviour (methods, showing what messages objects of class S understand and what code they will execute on receipt of them)

If class T inherits from S (e.g. `class T extends S`), objects of T automatically have this state and behaviour. [Warning: deliberate oversimplification!]

T may define new state and behaviour – extra data, and new methods, making objects of T understand messages that ordinary Ss don't.

T may also *override* S's behaviour, so that a T executes new, different code on receipt of a message that any S would understand.

Terminology so far

Conceptual relationships:

- T is a subtype of S / S is a supertype of T
- T specialises S / S generalises T

Implementation relationships:

- T inherits from S
- T is a subclass of S / S is a superclass of T
- T is a derived class of S / S is a base class of T (C++)
- T extends S (Java)

Dangers

Classes related by inheritance are very tightly coupled – any change to a superclass may affect all subclasses.

Overriding behaviour is especially fraught. The compiler can check the basics – e.g. does the subclass's version of a method return the same type as the superclass's? – *but it cannot check whether it is really a subtype.*

Inheriting an interface is much safer than inheriting implementation.

Take home

Don't use inheritance unless you need to.

Only use it where there is a subtyping relationship.

Adding *new* behaviour in a subclass is safe.

Overriding old behaviour needs great care.

Do not attempt to remove behaviour in a subclass.