Inheritance - problems with

There are some problems with inheritance

In our experience, inheritance isn’t always suitable

One problem with inheritance, is that it might be tempting to use it out of laziness “to get some stuff for free”™

You should realize that inheritance should be used with care, and only in situations where you truly express that you are writing a subtype which can take the place of any supertype.

An example we see a lot is that people are extending JFrame to get easy access to some methods, such as setLayout, setSize, setVisible etc.

We’ll see some problems with that next.

Common example

```
import java.awt.*;
import javax.swing.*;
public class JCompDemo extends JFrame{     // WHY???
  public JCompDemo(){
    JLabel top = new JLabel("Demo");
    add(top, BorderLayout.NORTH);
    JTextArea text = new JTextArea();
    add(text, BorderLayout.CENTER);
    pack();
    setSize(200,300);
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public static void main(String[] args){ // WHY main in same class??
    new JCompDemo();
  }
}
```

I need a window, so I'll extend JFrame, right?

Wrong.

If your class truly is firstly and mostly a new type of JFrame, then inheritance may be the way to go. If it isn't, you shouldn't use inheritance. Only inherit if your class "is a" supertype, not if it "has a" supertype or "needs a" supertype.

If your class needs to have a JFrame, it should have a JFrame variable, not express that it "is a JFrame".

What's the problem of being a JFrame?

First, you can only inherit one class, so if your class extends JFrame, it cannot extend any other class. The only reason for making this choice would be that most of all, your class is a special type of JFrame (to be used by other classes).

Second, if you extend JFrame, you get tons of methods inherited from the complete JFrame hierarchy. Except all methods declared in JFrame, you also inherit 163 methods from Component, 55 methods from Container, 76 methods from Window and 23 methods from Frame.

It is far from unlikely that you override one of those methods by mistake. And pretty darn likely that you don't need all that luggage!

It is hard to extend a class properly

There are many cases where you risk misinterpreting how a class works internally, and make assumptions which are not correct.

If you override one method, are you sure you understand the implications like what other methods is the original method calling?

Another problem is that if you extend a class and add a method to your subclass, how can you be sure that a future version of the superclass won't introduce a method with the same signature?

It’s not sure that we “save code” with inheritance

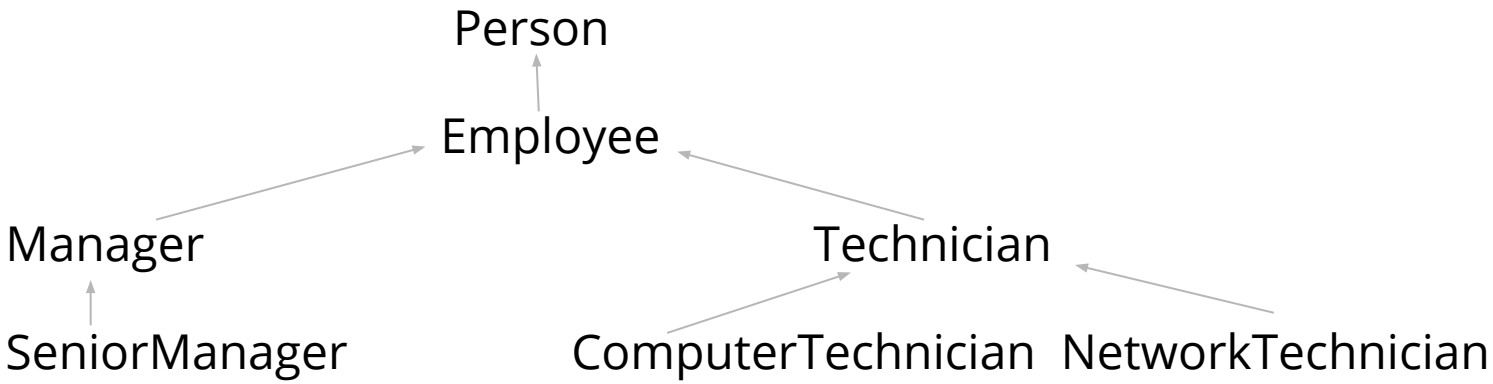
One often stated reason for embracing inheritance is that we can re-use code and don’t have to repeat ourselves.

This will however not always be the case. We’ve seen many examples of this code reuse, which supposedly should save us a lot of writing, where the result is a lot of classes which not only are very similar (e.g. no code reuse), but where some method or methods often needs to be overridden in order for the design to make sense.

It is quite possible to end up with a class hierarchy which has a lot of similar classes, all of which contain code for some method to be overridden.

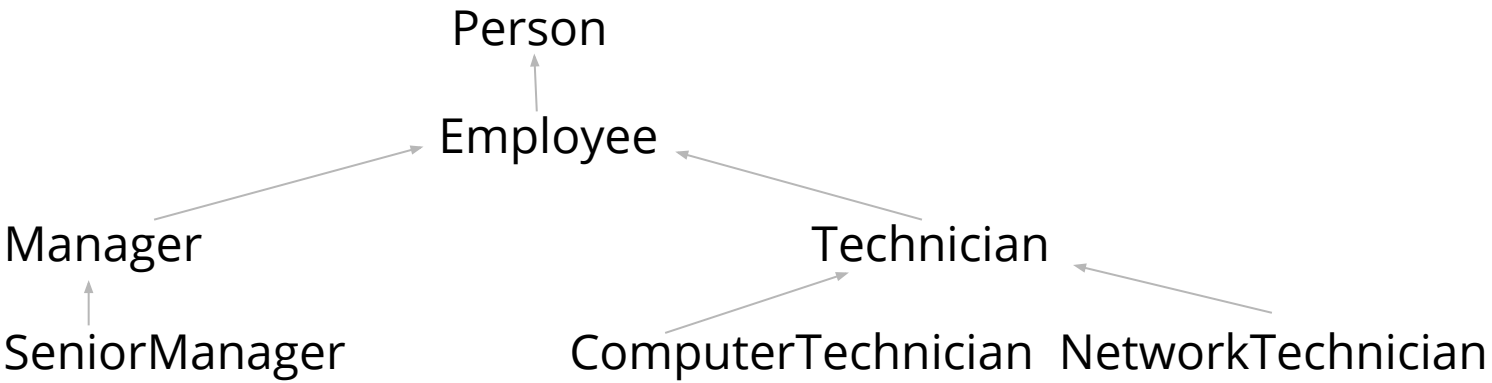
Overzealous inheriting may be inflexible

A common example of a class hierarchy goes something like this for a system managing the records of some company's employees:



Overzealous inheriting may be inflexible

Now, a record of all persons in the company can be represented as one of the types in the hierarchy, like for instance a NetworkTechnician.



But what happens if a ComputerTechnician is also a Manager of a team?

What could be done instead?

A different approach than using inheritance would to have simply one class, let's say Employee. An employee could have an attribute (an instance variable) with the role, or even roles!

This way, an employee could have both the role of a ComputerTechnician and that of a Mangager. Given the correct instance methods, the system could also handle that an employee gets new roles or quits some roles even in runtime. That would arguably be more flexible than the previous design where an object only could be of one class per branch.

Some words on protected

If you write a class and decide to let some methods and variables be protected, you force people using your class to use inheritance.

Remember, protected (no, we are not expecting you to use or know about protected in terms of the exam or assignments, this is just bonus information) means that the visibility is within the same class, the same package and subclasses.

A class with vital parts declared protected forces users to use inheritance to use said parts (unless they have access to the same package). Objects of such classes can’t be used as part of some other class (in another directory), because the vital parts are protected.

Some more words on protected

It could be argued that protected kind of breaks the idea of hiding information using private. If we subscribe to the idea that private creates a barrier between a class' internal workings and the public interface of the class, then we must consider that protected opens up internal stuff to classes in the same package and to subclasses.

If we need to expose some functionality to inheriting classes for flexibility, then we must choose between information hiding and flexibility.

Making the right call here is not very trivial.

Finding the right level of abstractions

Another common example when explaining inheritance is to look at the animal kingdom. Biologists have done a great job categorizing all living creatures into some kind of hierarchy.

What, we feel, is often forgotten when showing Java code for a class hierarchy of Mammals, Fishes, Crustaceans, Vertebrates, Birds, Reptiles etc etc in absurdum, is when would you actually need to model all those things in a computer program? For a simulation of the whole ecosystem?

Abstractions, in our opinion, should be about focusing on the stuff you really need in your system, and then only on the aspects relevant for your actual system. Not every noun is a class, nor do we need to model the whole universe.

Indications of the wrong level of abstraction

If you end up with a lot of quite similar classes, that certainly could indicate that you should reconsider your design.

Another indication would be the need to frequently write new classes which extend some superclass.

Going back to the Employee class hierarchy, if the company creates new positions a lot, and you have to add subclasses to the Employee class, that could indicate that you should consider some other approach, such as using a Roles object. If you can handle a new kind of role/position without having to write a new class, you probably have a more flexible design.