Privacy, Abstraction and Encapsulation

In this lecture we examine some important concepts in object-oriented programming.

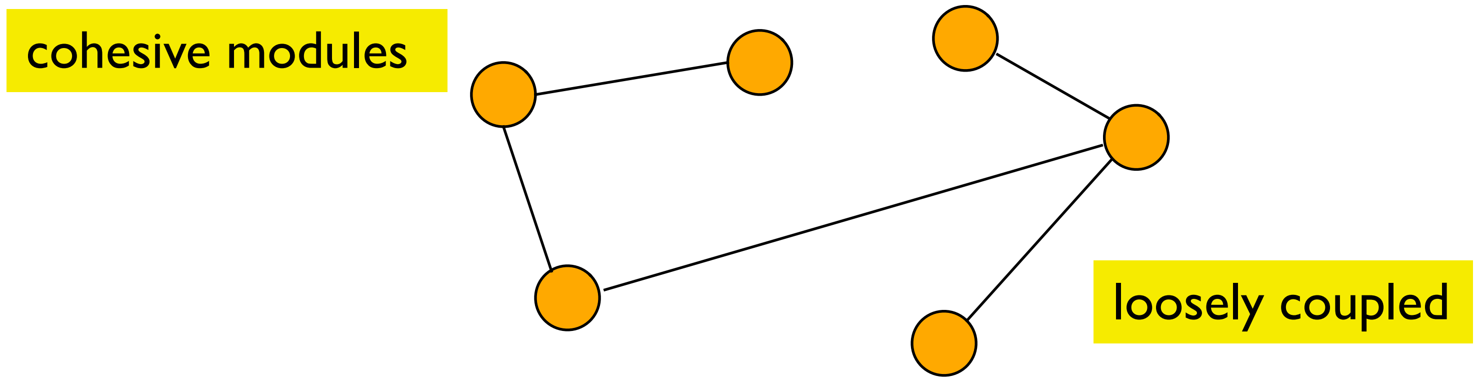
Most of what follows is applicable to OOP in general; some is Ruby/Java specific.

These topics will be covered in various places throughout any object-oriented programming textbook.

Why Private Data?

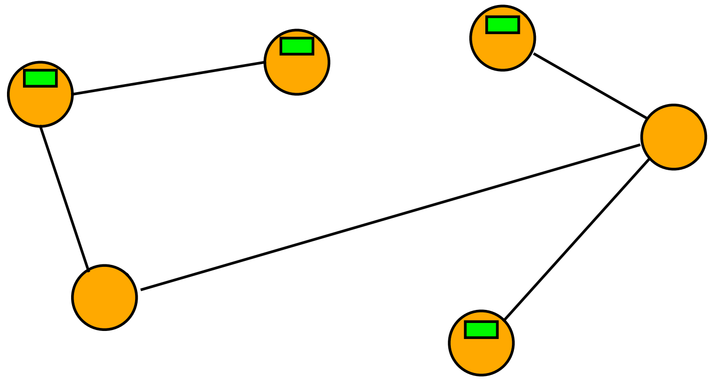
The importance of making the data members of a class private is well-known. Here we look at this in more detail.

The essence of any good systems development is the division of the system into separate, cohesive modules (classes) that interact in certain, fixed ways (message passing).



Duplication: the root of all software evil

Information that appears in several places around the system is contrary to this principle.



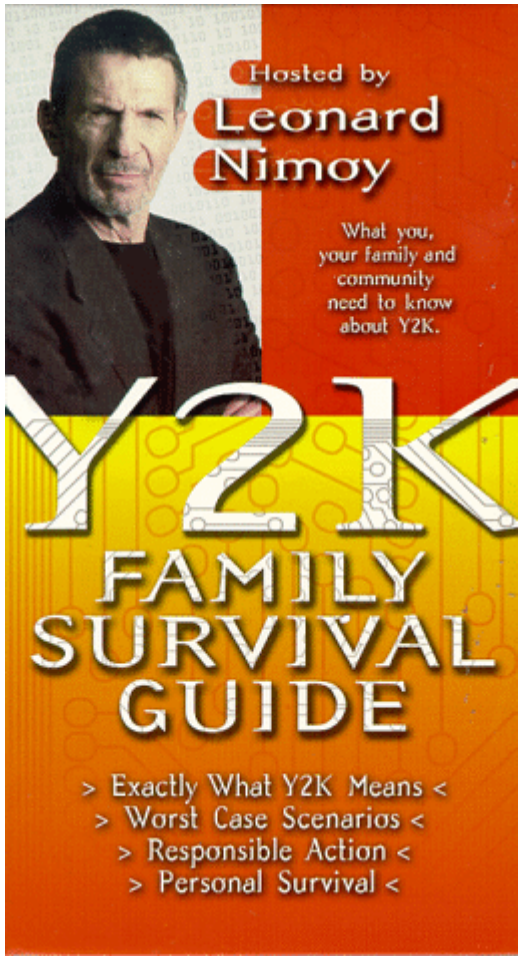
The green boxes above represent duplicated information; if one has to change, they all have to change.

The Year 2000 Problem

The Year 2000 problem (“Y2K bug”) was a good example of what can happen when this principle is violated.

The knowledge that the year was stored as 2 digits was spread across entire systems, so changing this to 4 digits was an enormous task.

(Predicted cost of fixing this problem worldwide were as high as $1 trillion; current estimates of actual cost incurred are more like $300 billion.)



Changing data format

Consider this simple Date class:

```
class Date
   attr_accessor :day, :month, :year

   def initialize day, month, year
      @day = day
      @month = month
      @year = year
   end

   def advance ndays
     ...
   end
end
```

Consider how to implement this method...

Consider updating this Date class to store the date in Julian form, i.e., as a single integer.

(Julian dates are widely used in astronomical software. They are expressed as the number of days since noon on January 1, 4713 BC. Today's date is 2455882.)

Data Hiding helps!

If we were using Java and had made the @day attribute public, this change would involve updating not only the Date class, but also every client of this class as well.

Because we use **data hiding**, client classes only have access to Date objects through the public interface of the Date class.

This means that we can make whatever changes to the Date class we want, once we keep the same public interface.

Clients will not know that the class has been changed. How could they?

Changing to Julian data format

Our new Date class would look something like:

```
class Date
   def initialize day, month, year
      @julian = # computation of day, month year
   end

   def day= d
      @julian = # ...
   end

   def day
      # ...
   end

   ...

   def advance ndays
     @julian += ndays
   end
end
```

These methods have to be written

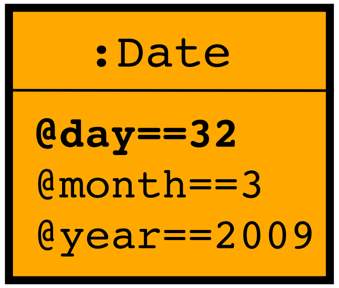
These methods becomes easier

The key is that **clients of the `Date` class are unaffected**.

Private Data helps debugging too

Another advantage of private data becomes apparent when debugging.

A large class of run-time errors occur when data takes on illegal values. For example, if the @day attribute of a Date object gets set to the value 32, the program is now in an erroneous state and an error will become apparent later.



Because the @day attribute in the original Date class is private, we know that if it must have been given its takes illegal value, in one of the mutator methods of the Date class.

Furthermore, the day= method can even be implemented to explicitly disallow any request from a client to change @day to an illegal value.

Abstraction and Encapsulation

This discussion of private data leads us to two important concepts in software development:

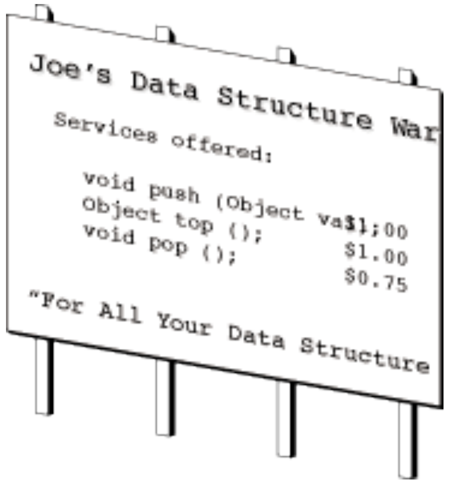
Abstraction
Encapsulation

These are closely related, but distinct concepts.

Beware! Different sources define these terms in different ways.

Abstraction

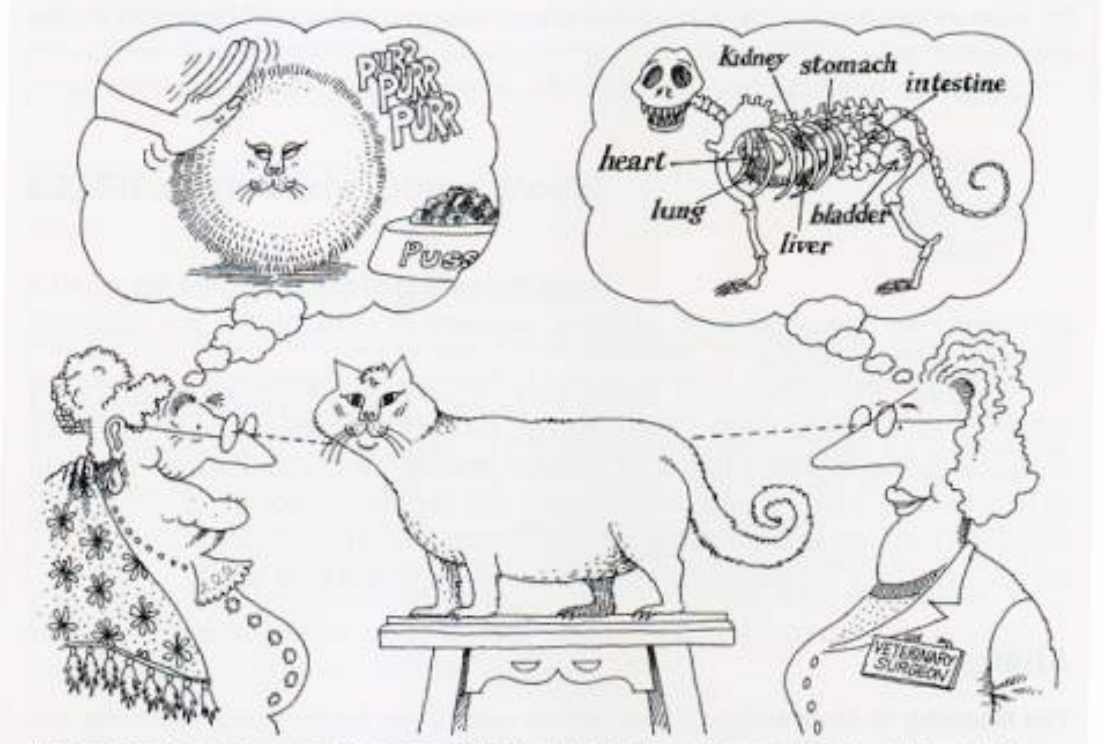
Abstraction is where the client of a class **doesn't need** know to know more than the interface to the class.



In the Date example, clients could happily use the Date class without having to know anything about how it was implemented internally.

Abstraction Perspectives

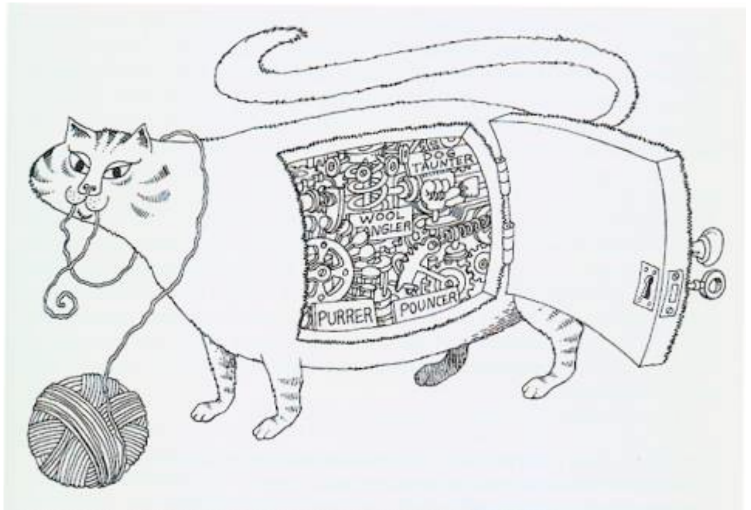
Abstraction is a relative notion; different clients may view the same object in different ways.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Encapsulation

Encapsulation is where the client of a class **cannot** write code that depends on anything other than the interface to the class.



Encapsulation hides the details of the implementation of an object.

In the Date example, the language semantics forbid a client from writing code that depends on the internal implementation of the class.

Abstraction and Encapsulation Examples

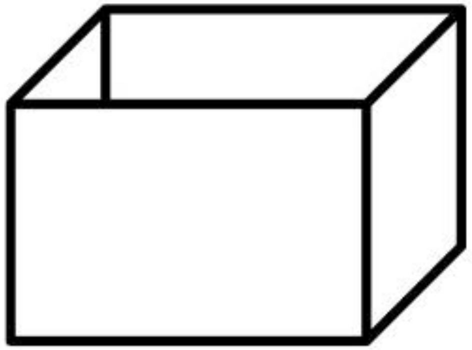
Abstraction and encapsulation are related but orthogonal concepts as the following examples show.

Comment on each of the following **Date** classes (written in Java, method implementations omitted).

```
class Date1{
   public int _day;
   public int _month;
   public int _year;
}
```

No encapsulation. No abstraction.

Clients can and must write code that depends on the implementation of this class.



Abstraction and Encapsulation Examples

Comment on this Date class.

```
class Date2{
   private int _day;
   private int _month;
   private int _year;
}
```

Perfect encapsulation! No abstraction though.

This sort of class looks useless, like a black box with no buttons on it.

Abstraction and Encapsulation Examples

Comment on this Date class.

```
class Date3 {
   public void advance(int nday){}
   public void print(){}
   public void set_month(int m){}
   public int month(){}
   ...
   public int _day;
   public int _month;
   public int _year;
}
```

Good abstraction! No encapsulation though.

A client can use the abstract interface to this class, but they can also write code that depends on the implementation.

Abstraction and Encapsulation Examples

How about this Date class?

```
class Date4{
  public void set_year(int y){}
  public void set_month(int m){}
  public void set_day(int d){}
  public int get_year(){}
  public int get_month(){}
  public int get_day(){}
  private int _day;
  private int _month;
  private int _year;
}
```

Encapsulation is perfect.

Abstraction is poor, but much better than just having public data.

Abstraction and Encapsulation Summary

Aim for both from the start.

Don't fall for the temptation of making data public thinking that you'll make it private later on. You won't; it's too much work.

Finally, the real value of these principles only becomes apparent when dealing with a large software project.

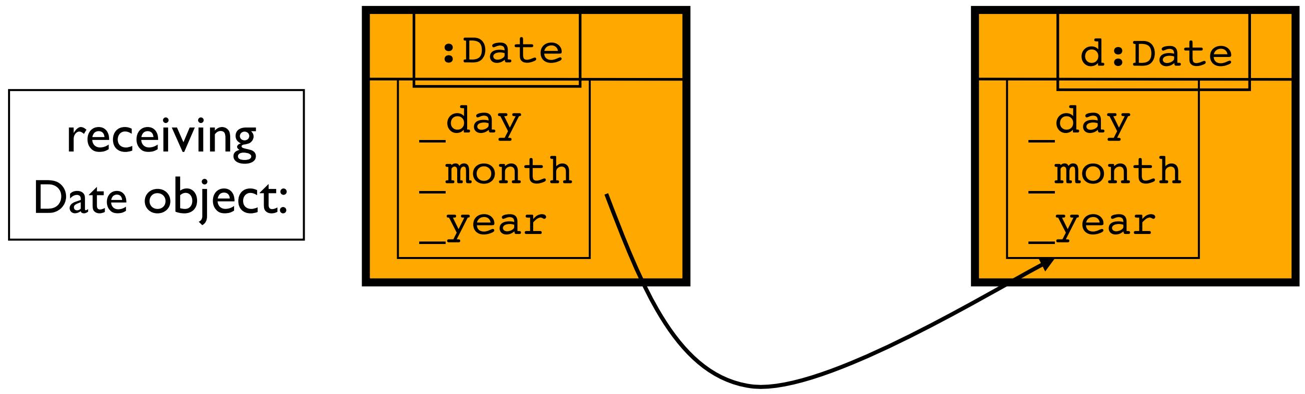
Wait! Not finished yet...

Privacy in OO Languages

Java Question: If we add this method to the Date class, will it compile?

```
int add_years(Date d){
   return(_year + d._year);
}
```

Conceptually what's happening is this:



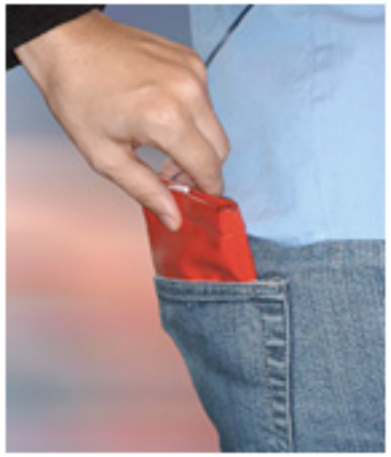
It looks wrong but…

Java/C++ Privacy is class-based

The privacy rules in Java/C++ are **class based**, not object based.

The above code is fine. The Date object that receives the add_years request can access the private data of d because d is also of the class Date.

Privacy in the real world isn't really like that
("here, have my wallet, we're both people after all")



$\xrightarrow{}$

leads to

How about in Ruby?

Ruby Question: If we add this method to the Date class, will it compile?

```
add_years(d)
   @year + d.@year
end
```

This won't work. Instance variables in Ruby are private to the object. Other objects, even of the same class, cannot access them.

A private method can't be accessed by an object of another class either.

Summary

In this section we looked at a number of more advanced issues involved in the design and implementation of classes.

Encapsulation and privacy guidelines are worth applying in your own programs.

However, the benefits of Abstraction are really only apparent when building large systems.