simply – OOP – simply

Let's talk about concepts.

I'll mainly talk about the **what is** and the **why** of OOP. The **how** – i. e. the syntax – is relatively easy to look up, when you've understood the concepts.

Image: http://www.sxc.hu/photo/998467

assono

IT-Consulting & Solutions

assono GmbH
http://www.assono.de
http://www.assono.de/blog

Thomas Bahn
tbahn@assono.de
+49/4307/900-401

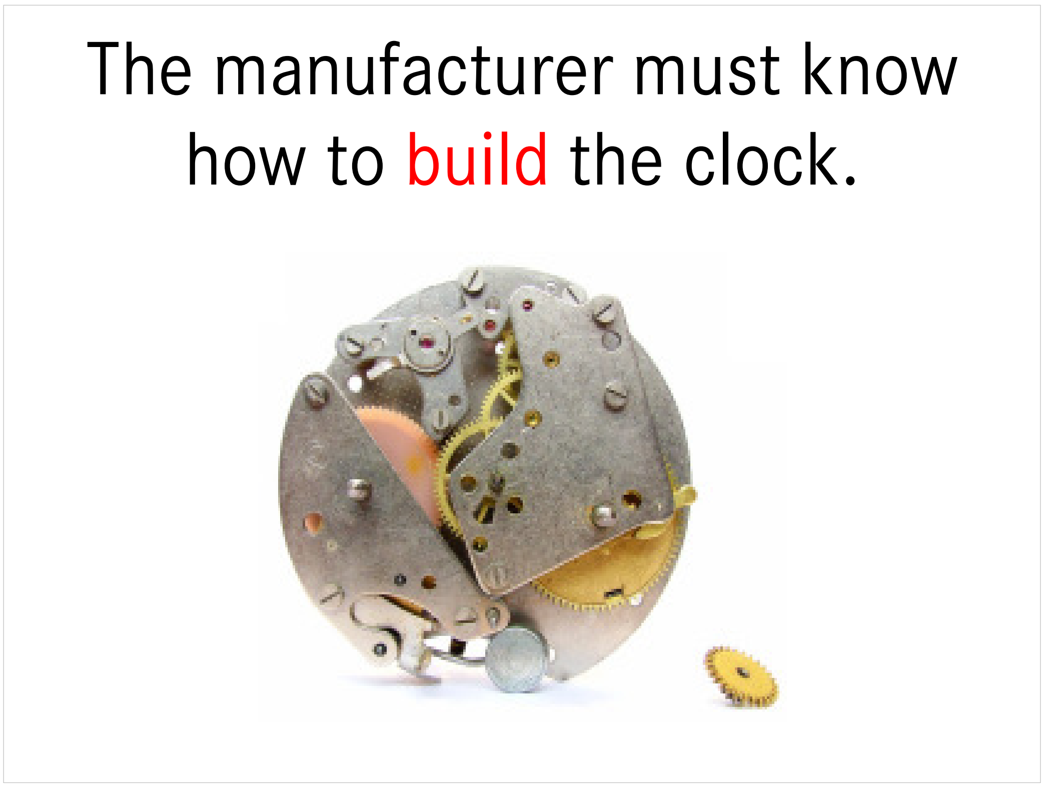
Thomas Bahn, 37
Graduated in mathematics (degree: Diplom-Mathematiker)
Cofounder of assono GmbH, an IT consulting company in
Germany
Developing in Java and RBDMS (OCP Admin) since 1997
IBM Lotus Notes and Domino professional since 1999
Development, Administration, Consulting, and Education
IBM Advanced Certified Application Development and System
Administration (R4 – 7)
Focus on interfaces to other systems (e.g., RDBMS, SAP R/3),
Domino Web applications, and GUI programming

You can use a pocket watch.

A watch has a very simple interface:

- hands to show the time
- a crown, which can be pulled out and/or turned either clockwise or anti-clockwise

Image: http://www.sxc.hu/photo/890003



The manufacturer must know how to build the clock.

The manufacturer must have detailed knowledge about the inner workings, the gears, the springs and so on.

Image: http://www.sxc.hu/photo/839239

A concept called information hiding.

Reduce complexity by hiding details beneath the surface.

All the complexity of the inner workings of the watch are hidden to the user. He doesn't need to now, how it works. He only needs to now the „interface“: how to read the hands and how to handle the crown.

The inner mechanics of the watch are hidden from him, thus not to confuse him.

The hidden, inner attributes and methods are called private, the visible ones are called public.

Image: http://www.sxc.hu/photo/696748

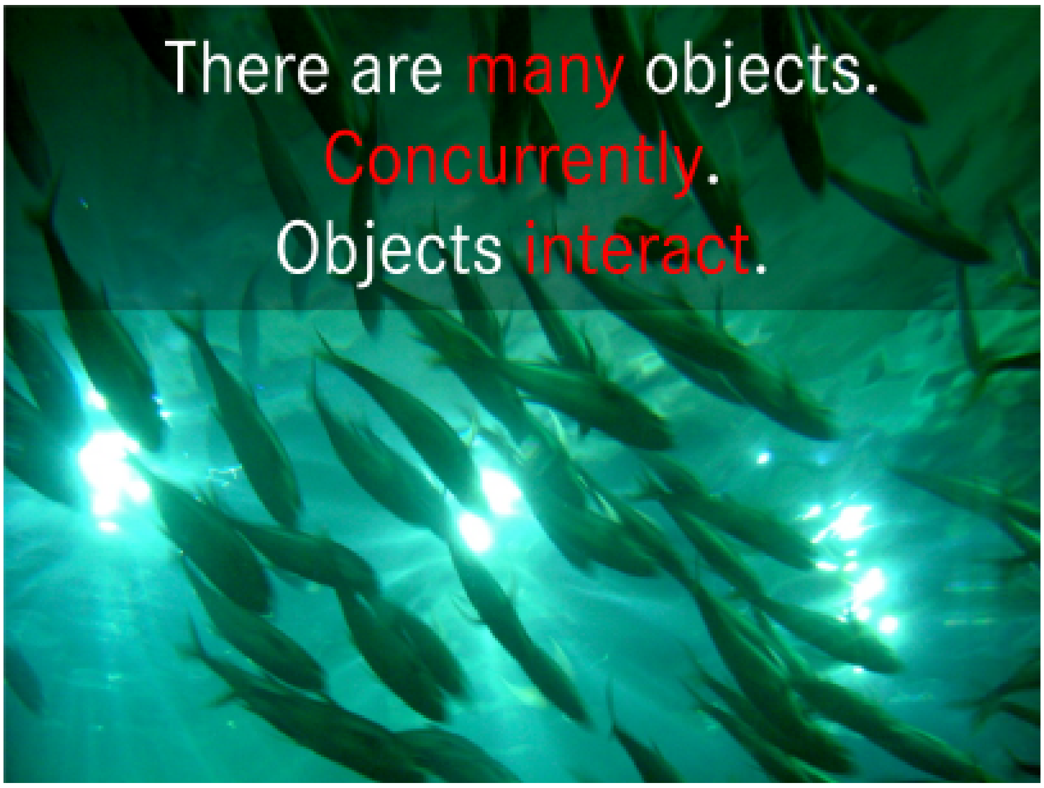
All private elements (details about „private“ follow) are hidden in a way, structured programs cannot do.
Script libraries can do something similar, although.

Image: http://www.sxc.hu/photo/982557

Everything is an object.
Everything consists of objects.

At least everything, we can see or touch.
Even humans are objects – in this abstract point of view.

Image: http://www.sxc.hu/photo/949574



There are many objects.

Concurrently.

Objects interact.

„Interacting“ in a quite wide sense of the word. Even if some paper lies on a desk, this could be considered as a kind of interaction. But most times, interactions are more „active“ than this.

Image: http://www.sxc.hu/photo/748081

OOP maps real world objects to software objects.

But even processes or relationships can be modelled as objects in programs.
Nonetheless, mostly programming is started by identifying by the real world objects to be modelled.
But OOP in Notes is more often than not about „abstract“ classes, like helper or tool classes, which aren't models of real world objects.

Images: http://www.sxc.hu/photo/999736, http://www.sxc.hu/photo/995000

A concept called encapsulation.

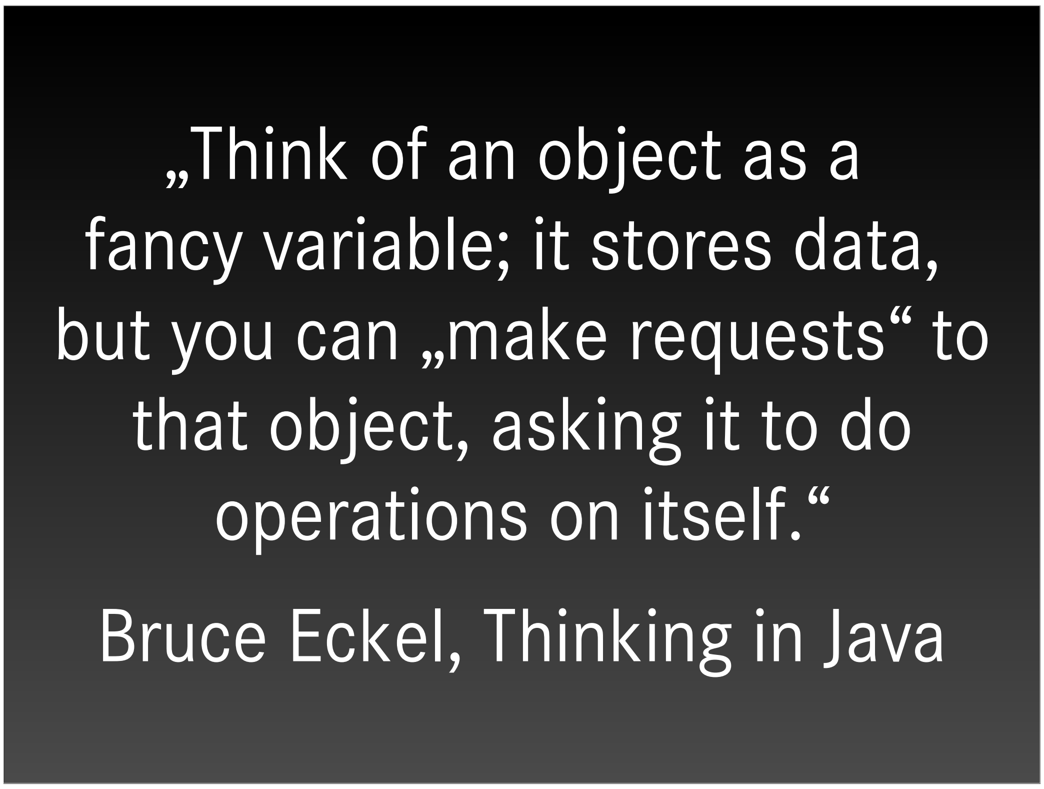
Object = state + behavior + identity

Comparison with structured programming:
different hierarchies for data and functions.
Data can be changed from everywhere, no „ownership".
OOP: Everything in one place, only the class' methods can change private attributes = state.

State = attributes = variables
Behaviour = methods = subs and functions

Identity means, you can always distinguish between two objects, even when they are of the same „kind" and have the same state – think of two watches showing the same time. They look alike, but are surely two different objects.

Image: http://www.sxc.hu/photo/1006559



„Think of an object as a fancy variable; it stores data, but you can „make requests“ to that object, asking it to do operations on itself.“

Bruce Eckel, Thinking in Java

Quote from 3rd Edition, page 33.

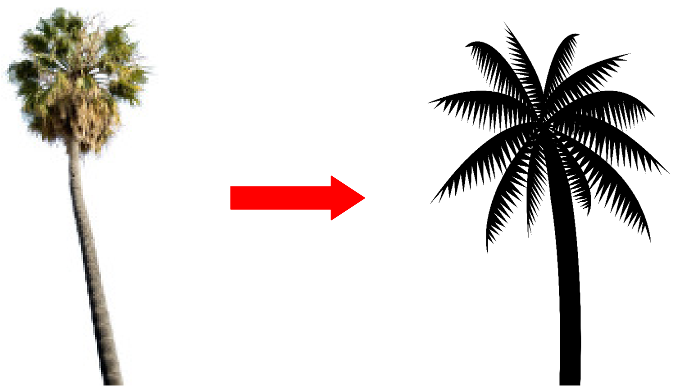
Models of an office chair

When a manufacturer thinks about an office chair (e. g. for his catalog), he might think about height, weight, materials, features etc.
When a book-keeper thinks about an office chair, he might think about price paid, date purchased, date delivered, expected time of usage etc.
When a moving company thinks about an office chair, it might think about the dimensions, how much space it needs in the truck, is it possible to take it apart etc.

There is not one model of an real world object, but many. It all depends on the context.

Image: http://www.sxc.hu/photo/588853

A concept called abstraction.



Reduce complexity by ignoring unnecessary details.

In any given context, only some of the many characteristics of an object are necessary.

To reduce complexity, models should only those details necessary for the intended purpose. By removing unnecessary details, complexity is reduced.

Images: http://www.sxc.hu/photo/996245, http://www.sxc.hu/photo/999574

Only abstraction makes real world objects manageable for programs.

Image: http://www.sxc.hu/photo/974531

A concept called class.

A class is like a stamp or blue print to create objects.
Each watch is an object; the construction plan of the manufacturer is the class, defining the possible states (attributes) and behaviour (methods).
In programming, the process of creating objects from classes is called instantiation.

Image: http://www.sxc.hu/photo/770543

Divide and conquer: Software design class for class

Programming in OOP is a (at the very least) two step process:

- Identify the objects and define the classes needed
- Implement each class – one by one

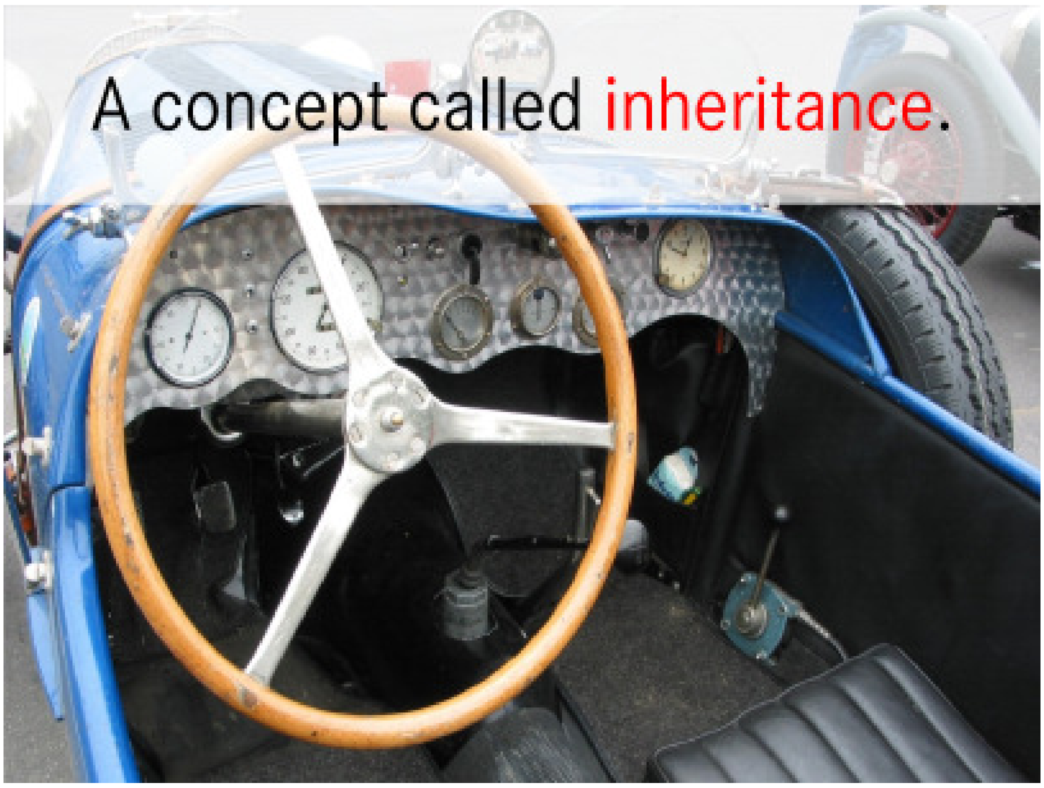
This is the classic „divide and conquer“ paradigm.

Image: http://www.sxc.hu/photo/443042

This is the power of OOP.

By dividing the development process this way, the programmer can either concentrate on the „world view“ of the system, where internal details of the classes aren't visible, or on the „microscope view“ of one class (at a time), when implementing it.

Image: http://www.sxc.hu/photo/343423



A concept called inheritance.

All cars have brakes and and accelerator pedal, a steering wheel and some instruments showing some details, like the current speed.
They have a lot of attributes and behaviour in common.
But then there are some „special“ kind of cars, like trucks. They still have the „common“ car attributes and methods, and some additional ones, like load capacity, the kind of load (liquids, containers, cars :-)).

In OOP you can build a class for cars with all the common attributes and methods. Then you can create another class, e. g. Truck, and say: a Truck is-a Car, thus has all of Car's attributes and methods, and add some additional variables, subs and funcitons.
This is called inheritance. In this example, Truck is a subclass of Car and Car is Truck's superclass.

By the way, a variable of type Car can contain Car objects (obviously), but also Truck objects or objects of any subclass of Car!

Image: http://www.sxc.hu/photo/396573

You can imaging, how much code you can reuse just by declaring one class a subclass of another one. Simply all the the superclass' code!

In the first demo I will show a class Watch and a subclass WatchWithSeconds.

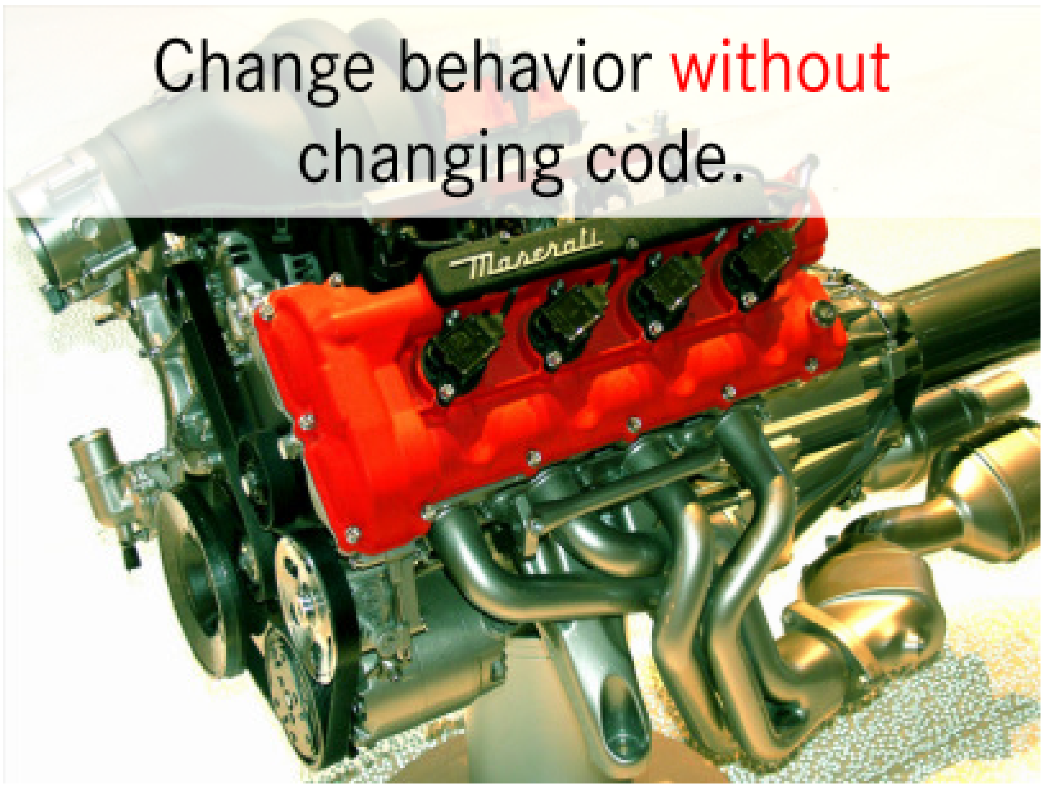
Image: http://www.sxc.hu/photo/789140

A concept called polymorphism.

Subclasses can have additional characteristics and behavior. But they can also have a different behavior than their superclass, like the racing car might have another acceleration method than a normal car.

When both classes, Car and RacingCar have a method with the same name, e. g. Accelerate, and you call this method on an object, the program decides at runtime which method is executed. If the object is of type Car, the Car's mehtod is called, if it is a RacingCar, RacingCar's sub is executed.

Image: http://www.sxc.hu/photo/297733



Adding a method to a subclass with a method of the same name in its superclass „overwrites“ this method.

Consider this example: You have developed a class Engine. It has been tested and is in your application for a long time.

Now you need something similar, but it must have some other behavior, like a different acceleration. You can write a subclass RacingCarEngine of Engine, with only one new method: Accelerate.

You have a variable of type Engine in your application. You only assign it a RacingCarEngine instead of a „normal“ Engine object.

Everytime an attribute or methode besides Accelerate is used, its Engine's attribute or method. But when Accelerate is called, RacingCarEngine's sub is executed.

This way you have changed the behavior (another sub is executed) without changing one line of Engine's code!

Image: http://www.sxc.hu/photo/206507

This way, you really know, Engine doesn't need to be tested, since it hasn't changed at all! You can concentrate your testing only on the new code, i. e. the RacingCarEngine class.

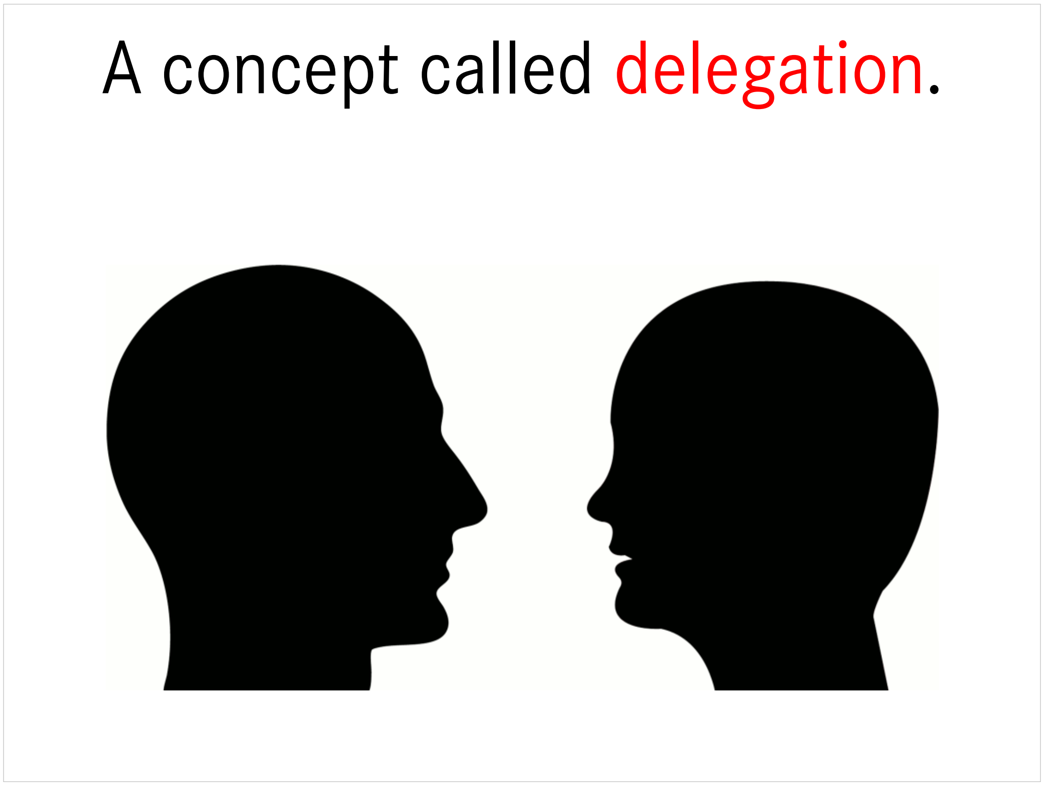
This is real code reuse!

Image: http://www.sxc.hu/photo/289829

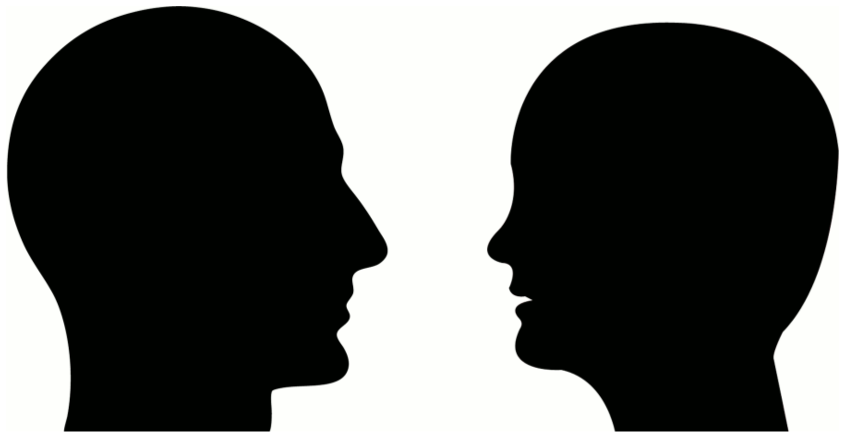
Everyone has his responsibilities. Every object, too.

In the case of an car, the engine has the resposibility of accelerating the car.

Image: http://www.sxc.hu/photo/321961



A concept called delegation.



The engine is a part of a car. Or the car **has an** engine. It uses its engine. It delegates the task of acceleration to the engine.

A class can have an variable (attribute) of a class type, that is, has objects as values. Then this class can call this objects methods to fulfill a specific task. It delegates some of its work.

Image: http://www.sxc.hu/photo/980826

This is the power of OOP.

By attributes of an object type you can divide a complex system like a car into simpler subsystems. Again **divide and conquer** of a problem.

By delegating work to an inner object, you can also change behavior by simply exchanging the object stored in the variable by another one. Think of replacing the engine of a car with a new, stronger one.

As you can replace the engine long after the car has been built, you can exchange the object with another one at the runtime of your program. That means, you can change the behaviour of the Car class at runtime!

This makes delegation even „better“ than inheritance, where you can change behaviour at (before) „compile-time“.

Image: http://www.sxc.hu/photo/722800

Construction and destruction of objects

When you use a class to create a new object, a special method is executed. This method is called the constructor of the class. It can be used to initialize the state of the object that is the content of the attributes, or you can open files or connections to other systems.
As any other method, the constructor might have arguments to initialize the class' state.

The opposite of constrution is destruction. And so the there is an opposite of a construtor, too: the destructor. This method is called, before an object is destroyed. This can happen, when the object is deleted or when it is stored in a local variable and the containing code block is exited. You might use a destructor to free reserved resources, like closing an openend file or connection.

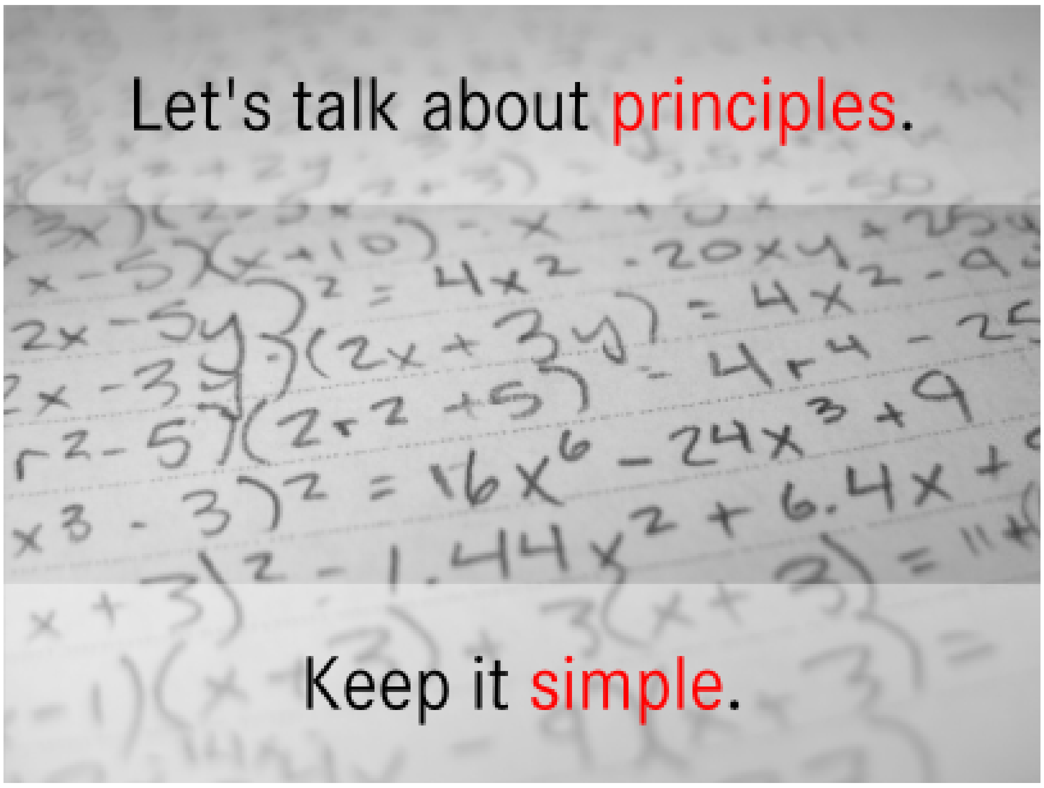
In LotusScript the constructor is called

    Public Sub New

and the destructor is

    Public Sub Delete

Image: http://www.sxc.hu/photo/600354

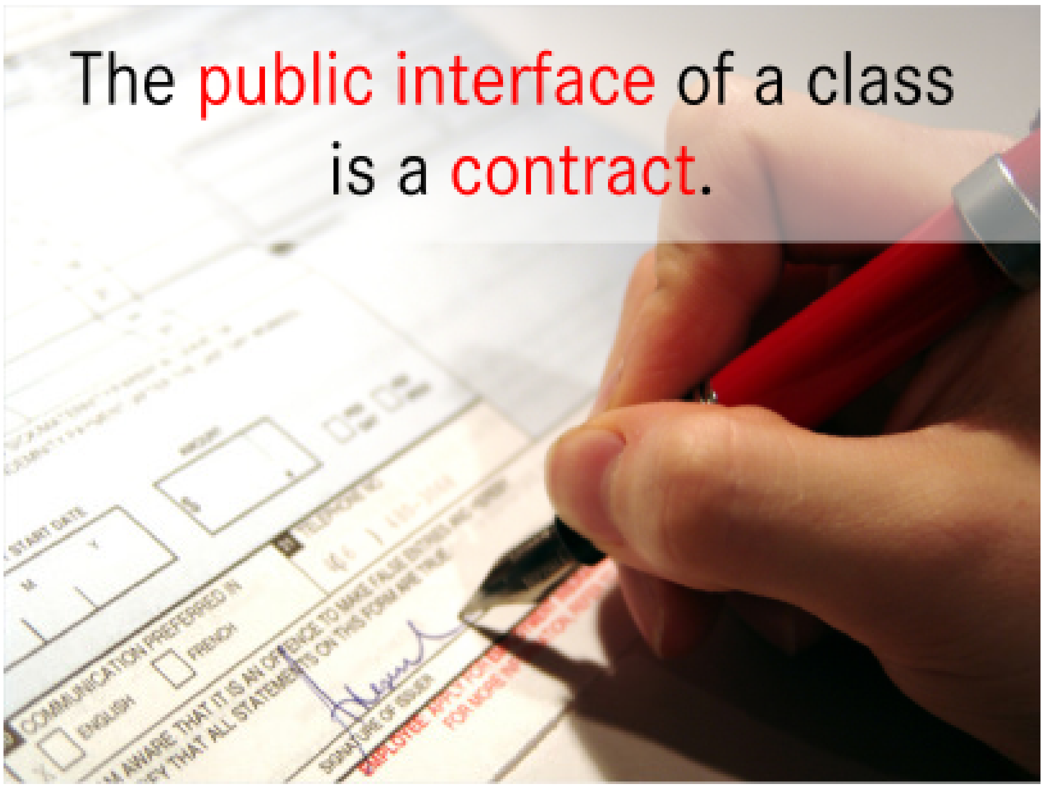

Let's talk about principles.
Keep it simple.

Writing your applications with classes doesn't make them „better“ out of itself.

You should heed some basic, common sense principles to get the most out of OOP.

The most important principle is to keep your programs as simple as possible while still fulfilling their task. Making them more complex than strictly necessary just increases development and testing time, decreases maintainability and readbility.

Image: http://www.sxc.hu/photo/614682



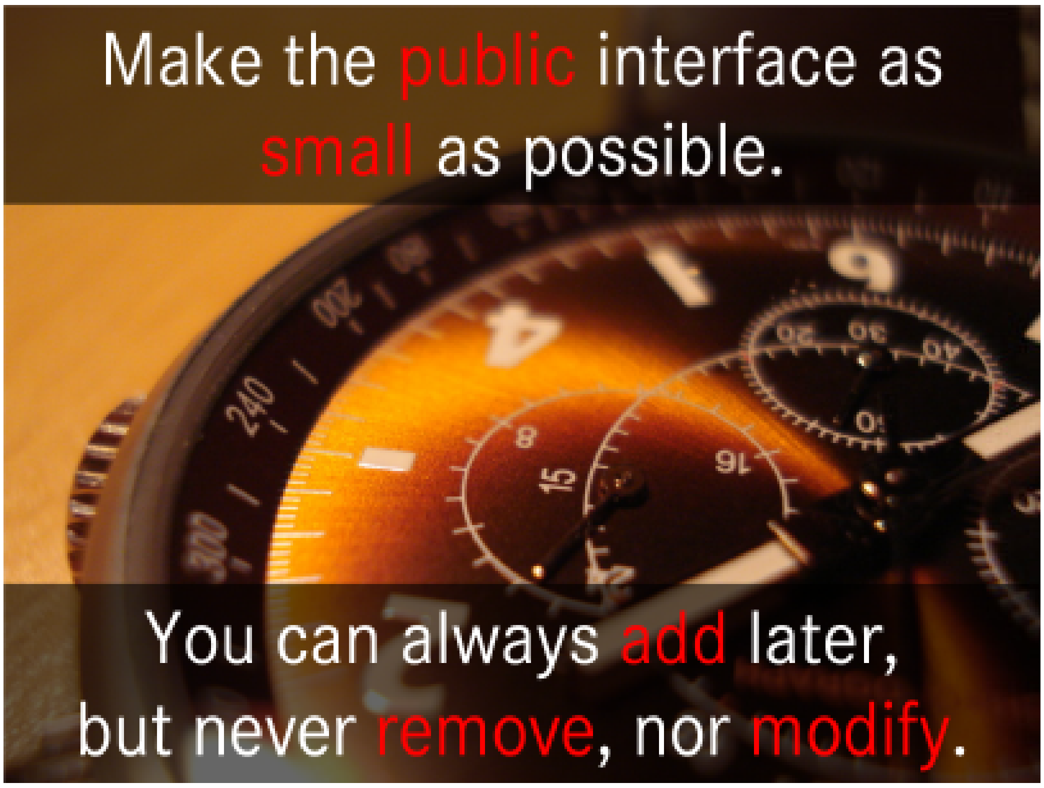
The public interface of a class is a contract.

The public attributes and methods of an class have a name, parameters and a given semantic, often described by the name and some comments.

The „user“ of the class can and must rely on this characteristics to use the class.

Just like a contract, you must fulfill this „promise“.

Image: http://www.sxc.hu/photo/204756



Make the public interface as small as possible.

You can always add later, but never remove, nor modify.

When you remove or modify an public attribute or method, e. g. change its name, some code using your class can become invalid. Thus you must notify all users of your class and they must change their code. Then all changes must be tested, again!

Adding new members to your class cannot have this consequence.

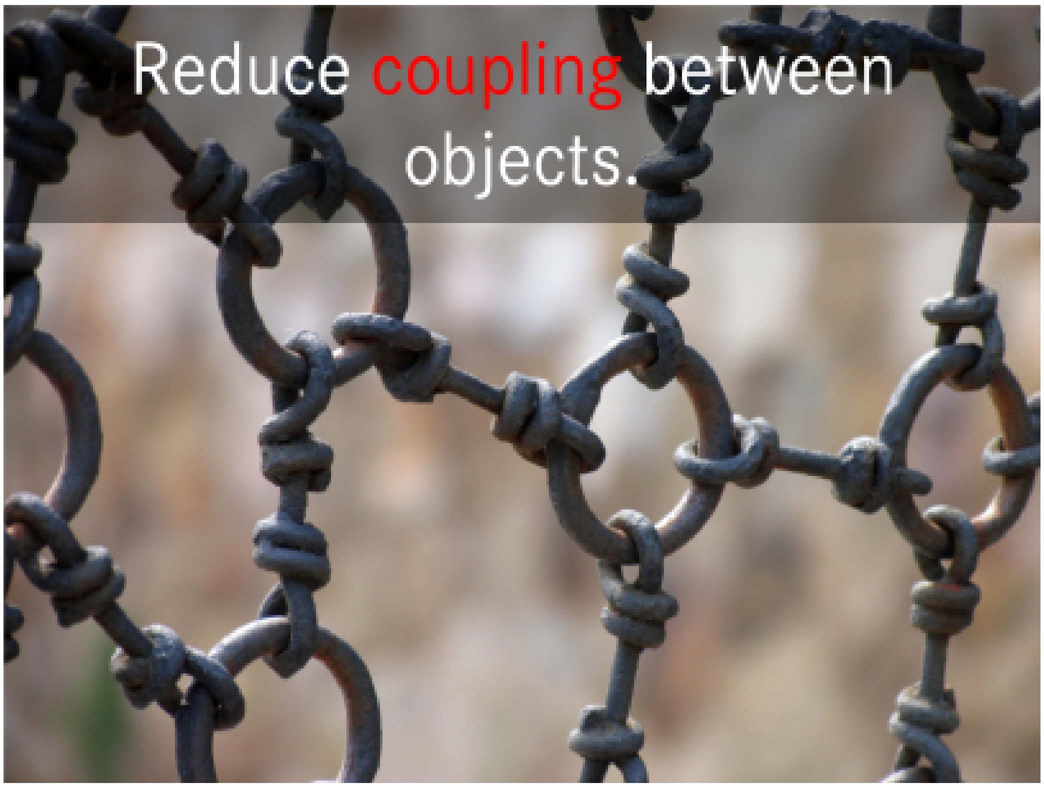
By making the public interface as small as possible, that is, making all members possible private, you reduce the propablility of a need to change a public attribute or method.

Image: http://www.sxc.hu/photo/841825

Remember: You can always change the private internals .

But you can always change the inner workings, like the watch manufacturer can exchange the spring operated clockwork by an battery powered one.

Image: http://www.sxc.hu/photo/921108



Reduce coupling between objects.

Coupling of objects mean their dependencies. When an object calls a method of another one or uses one of its attribute, it depends on this other class.

Reducing coupling means reducing of those utilizations of other objects – mostly by calling lesser objects, not making lesser calls to other objects.

Having a lot of dependencies between your objects make your program inflexible like this metal fence. When you try to move one element, other will most propably be effected. Or in OOP lingo: changing a class can effect depending classes.

Image: http://www.sxc.hu/photo/711284

Test your classes separately first.
A concept called Unit Testing.

Like always, test early, test often.

Classes are an ideal unit of testing, because they are small components and should be quite secluded.

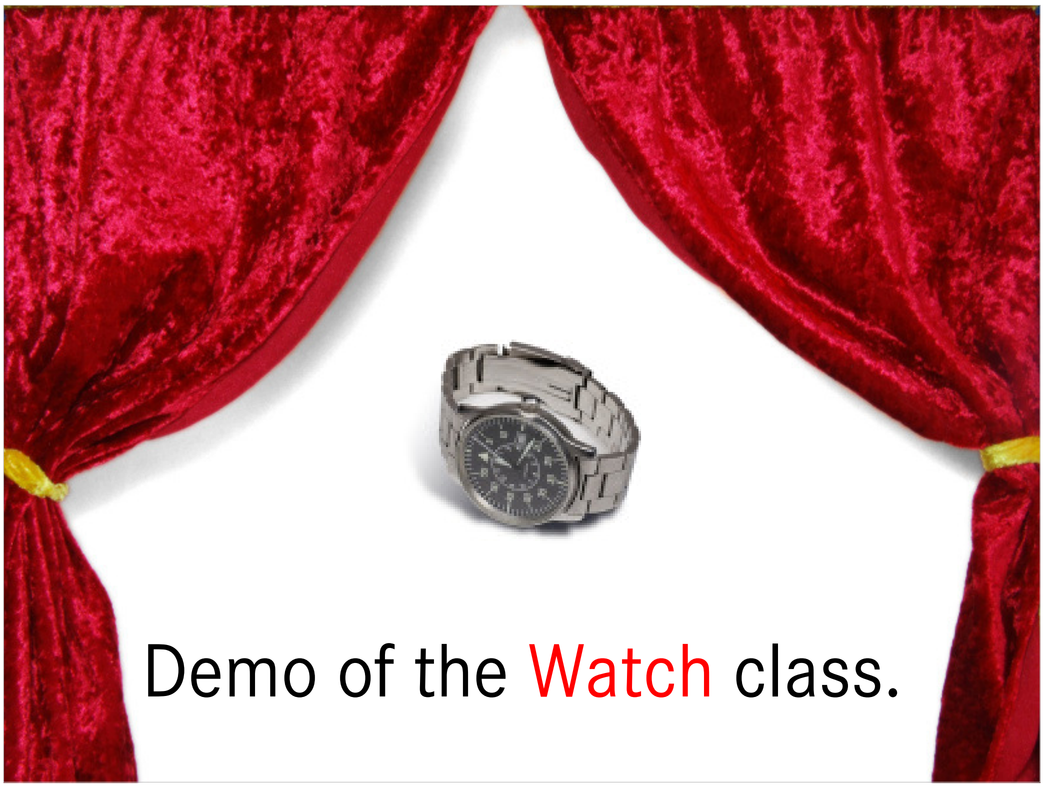
After you have tested each class seperately, integrate them to a bigger component and test again. Repeat until you have tested the whole program.

Image: http://www.sxc.hu/photo/213004

Let's talk about OOP in LotusScript.

I won't tell you all the syntax in LotusScript. Just use the Designer' help to get a description of the exact names, sequences, parameters etc.

Image: http://www.sxc.hu/photo/948591



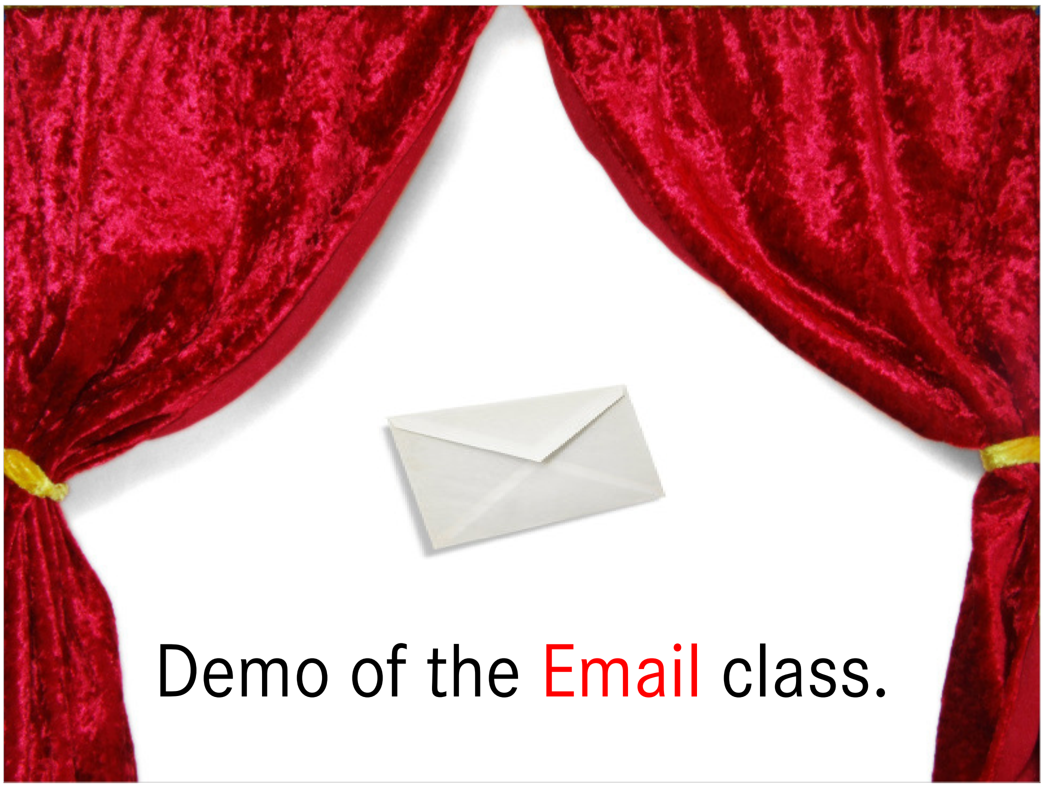
Have a look at the Watch class (in Class Watch script library of the demo database) and its subclass WatchWithSeconds. The agent Demo Watch Classes contains some test code and shows, how to use these classes.

All attributes are private.
Look at the constructor (New) and destructor (Delete) of Watch.
There are three internal = private methods.
TickTack should be called once a minute, but NotesTimer doesn't work in agents.
PushKnob and PullKnob change state of the knob (crown).
TurnKnob only uses internal methods (depending on state and arguments).
GetTimeAsString calculates a formatted string.
PrintTime prints this string, the current time of the watch.

The class WatchWithSeconds only has one attribute and 4 methods, but reuses 4 attributes and 7 methods of its superclass – Watch.
Its constructor has one more argument and calls Watch's constructor first.
I only overwrite some private methods, but leave all public ones unchanged.

It's absolutely simple to use these classes.
In the for loop, I call TickTack 15 times, although the watch has only „energy“ for 10 minutes.
Watch the output!

Images: http://www.sxc.hu/photo/607060, http://www.sxc.hu/photo/955323



Have a look at the Email class (in Class Email script library of the demo database) . The agent Demo Email Class contains some test code and shows, how to use this class.

For the comments I use the ls.doc comment convention. See http://www.lsdoc.org/
This way, ls.doc can extract this comments and create a „web site“ of them (just like javadoc).

We had a sub to send emails (like @SendMail) with SendTo, Subject and Message as parameters. Then we added CopyTo and BlindCopyTo. Then a Doclink with comment. Then we needed more granular access to the Body RichText item to add styles and more than one doclink. Then we wanted to set the Sender and Principal. We got a lot of different variants of SendMail sub with different arguments (and quite long names).

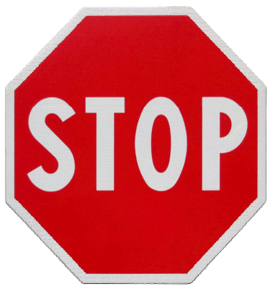
Then we created this class to handle all of our requirements.

Afterwards we added the ability to add more items to the mail send. Therefore we simply had to add an attribute and some public methods (and a minimal change to the Send mehtod). This was a lot easier than it would be to add this functionality to many different SendMail subs we had before.

We could add some „convenience“ subs as public subs to the script library, which uses our class to send mail. These subs could have the most typical combinations of arguments and would allow to send mails in just one line of code (again). But since „users“, i.e. other programmers could still use the class itself, if necessary.

Images: http://www.sxc.hu/photo/607060, http://www.sxc.hu/photo/890671

The end



Ask questions now – or later

Blog:	www.assono.de/blog
Email:	tbahn@assono.de
Phone:	+49/4307/900-401

Image: http://www.sxc.hu/photo/200984