

# CS209

## Computer system design and application



Stéphane Faroult

## ABOUT ME

Office: Room 914, Building A7, 9<sup>th</sup> floor  
iPark, N° 1001 Xueyuan Road

Office Hours: Tuesday 14:00-16:00  
Thursday 16:00-17:30  
or send me an email

Stéphane Faroult  
[faroult@sustc.edu.cn](mailto:faroult@sustc.edu.cn)

Here are my coordinates. I'll be there during office hours but I'm also most time in my office when I'm not lecturing or teaching labs. If you need explanations and cannot come during regular office hours, drop me an email and we'll find a slot.

## Teaching Assistant

Office: Room 913, Building A7, 9<sup>th</sup> floor  
iPark, N° 1001 Xueyuan Road

赵耀(ZHAO Yao)  
[zhaoy6@sustc.edu.cn](mailto:zhaoy6@sustc.edu.cn)

If there is nobody in room 914, you can try the next door ...

## About this Course

Get a deeper understanding of programming

Learn many advanced Java features

Discover new topics that you may have  
approached on your own (or not)

Graphical Interfaces

Web-application backend

Android applications (*if time permits*)

## About this Course

I supply annotated slides (PDF) before the next lecture.



Of course, if you are reading this, you have already found them. I also supply an index to notes (starting from Lecture 2)

## Textbook

Only what I think is important.



A few important topics are absent from the book.

The textbook is the same as for Java 1, but I'll follow it only loosely. I'll update it where it's a bit obsolete (although what it tells is still for the most part very much in use).

You may find explanations better in the course notes on some topics, and better in the book on some other topics. Hopefully, one will help you understand the other better.



Different views of the same thing.

If you really want to get the most out of this course, read the notes from the preceding lecture at least once before a lecture, and also check, not necessarily in deep detail, the book. Notes are always provided AFTER the lecture, but you may want to have a quick look at the book before (or after) the lecture.

Read the notes regularly.

Check the book before or after the lecture – whatever works best for you.

You'll find at the end of the notes indications about the topics that will be covered next and the corresponding chapters (if available) in the book.

### Grading

**LABS** 40% of the grade

MidTerm + Final Exam 20% + 30%

A few quizzes 10%

I'm a strong believer in practice and want to ensure that you can code. If you do well in labs you'll probably pass even with exam grades that aren't stellar.

### All exams are



Open book, open notes



Electronic devices forbidden

Because professional developers have access to everything, you can use books and notes in all exams. Electronic devices are forbidden to eliminate the temptation of asking a friend.

### CREATE YOUR OWN NOTES FROM MY NOTES

(or the book)

Because these notes are electronically distributed, you'll have either to print them (lot of printing) or (better) create your own notes, which is an excellent way of learning.



### Warning!


if you get 50% at an exam, it's OK

My raw marks are tougher than what you are accustomed to; if I give you 50%, it's a passing grade, and I may decide that somebody with less than 50% can pass. I curve grades to fit the commonly used scales before grades are entered in the system. I tell after the MidTerm exam what I consider OK and not OK so that you know where you stand.

## A review of Object Orientation

Let start with what "Object Orientation" is really about. When you start with programming, you are really concerned about "if" statements and loops, you are happy with having a program that compiles and gives the expected result, but usually you don't go much further. With experience, you realize that you may have many ways to organize a program. The first programming languages were purely "procedural" languages, describing steps to perform to obtain the result. Object orientation brought a new way of looking at things. There are still other ways and other kind of languages, such as declarative languages where you simply state what you want (database queries) and functional languages. Depending on the problem to solve, one approach may be easier than another one.

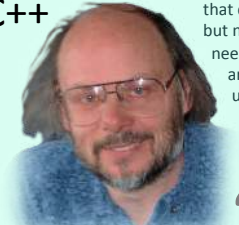
**Ole-Johan Dahl** (1931-2002) **Kristen Nygaard** (1926-2002)



Object Oriented concepts were invented by two Norwegians who were interested in using computers for running simulations and created a dedicated language in the 1960s.

About 15 years later a Dane, Stroustrup, implemented their ideas to improve C, which had been created in the meantime and had become very successful and popular. C++ also became popular, but it's a language that does some things that C doesn't for you but not everything, and, as with C, you really need to know what you are doing. C and C++ are reputed difficult, but are still widely used because they run fast.

**C++**




**"C with classes"**

**Bjarne Stroustrup** 1950-

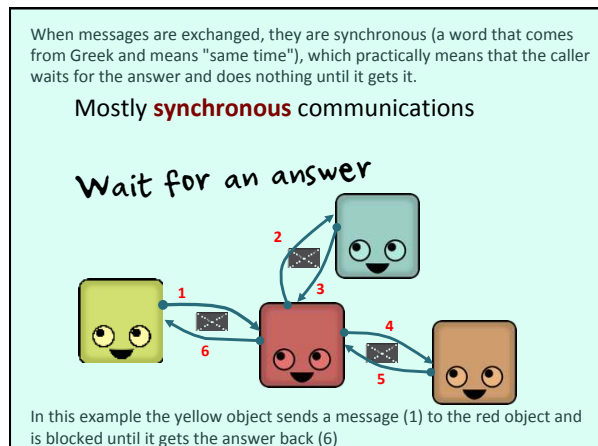
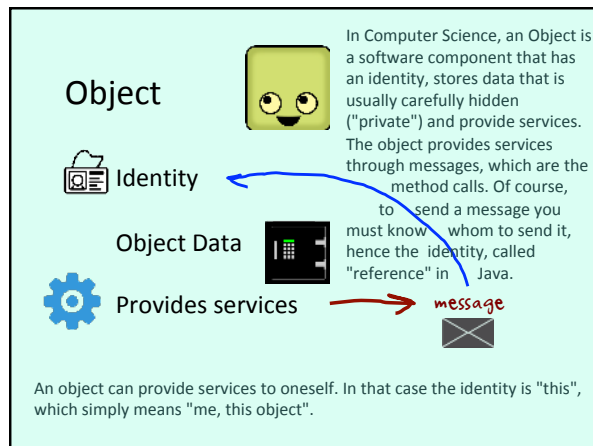
**"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."**

**"Java is C++ without the guns, knives, and clubs"**



After another 10 years (we are now in the 1990s), James Gosling came out with Java, which draws heavily on C and C++, but is far simpler and automates memory operations. Most notably, it can run in different operating systems very easily. Its architecture (the virtual machine, coming soon) sets it apart from C and C++. Java is a little slower than C and C++, but not much, and is today one of the main languages of business applications.

**James Gosling** 1956-



When many objects exchange messages it's said that there is "coupling" and it can be very slow (especially when, as it happens sometimes, the objects aren't all running in the same machine). There is a bit of a dilemma here, as it's usually better to divide a big task in many elementary tasks executed by different objects, but then it increases the number of messages to exchange.

**Coupling**  
Can be slooow

I have said early that you need the identity (reference) of an object to exchange a message (call a method) with it. An object can create another object by invoking its "constructor" and the constructor returns the reference. But the creator may not be the only object that expects services from the newly created object. In many cases, it will pass around the reference (as a method parameter) so that the other objects it requests a service from can use services from this new object. It's not recommended to let every object create objects, and it's considered to be a better practice to have objects that act as "object factories" and then pass along references. This is a technique known as "dependency injection".

**An object can create other objects**

**A creator gets a reference to created objects**

**References are data**

**References can be exchanged through messages**

When you look at an object application, you find that, at least on the paper, it's something very simple: a set of objects, all specialized in providing a simple service, that work together and exchange messages to provide the global service that the user expects from the application. The difficulty is of course in implementation, how you slice everything, and how you can reuse existing pieces of software.

## Object Application

Set of objects

that communicate

to provide a service to the application user

As with chickens and eggs (which one came first?) there is a boot strapping problem, in other words how you start the process. In Java, there is a very special object with a method called `main()` that pops out of nowhere to create the first objects in the application and start everything running.

## Boot strapping



Special object (*main*) creating the first objects.

As I have already hinted, one of the problems in object oriented programming is that some of recommended strategies are at odds with each other. Small objects doing small tasks are better for consistency (which means that objects are focused on one task), reuse, and testing (you can write a mock object that returns hard-coded values while someone is debugging the real thing); but they increase communications and coupling.

Better to have several small objects than a big one (testing) - consistency

Better to pass references than leave *any* object create new objects

Better to minimize communications (weak coupling)

You mustn't forget some of the key principles of object-oriented programming: encapsulation, which means that everything is private except the public methods that define the interface of the object with the outside world, and responsibility.

## Key Principles

Encapsulation



Responsibility – the object has all the means to provide the service

in theory, Object Oriented programming is easy. There is a saying that "the gap between theory and practice is wider in practice than it is in theory". One (but perhaps no the greatest) difficulty is to correctly identify objects. In spite of their name, "objects" don't always correspond to real-life objects; sometimes they implement a function. Let's illustrate it with an example.

Object Oriented programming is all about

- independent items
- having a status
- having a behaviour
- exchanging messages

If you work for a lift (elevator if you prefer American English) manufacturer, you may have to code software for the operation of lifts. Optimizing the movements of lifts so as to minimize waiting times is not a small task, knowing how many lifts are required in an office building depending on the number of people working in this building can also be challenging, especially as people tend to massively use the lifts at the same time. You may need to run simulations, which takes us back to the roots of Object-Oriented programming.



We may think of creating a lift class. First of all, it must know which floors it serves. Then, it has a state: it may be stopped (for maintenance sometimes) or moving, up or down, and won't change direction before it has reached an "extreme stop". Messages are of two kinds: an "up" or "down" call from the outside (floor is known) and a "get me to floor" call when people press a button inside.

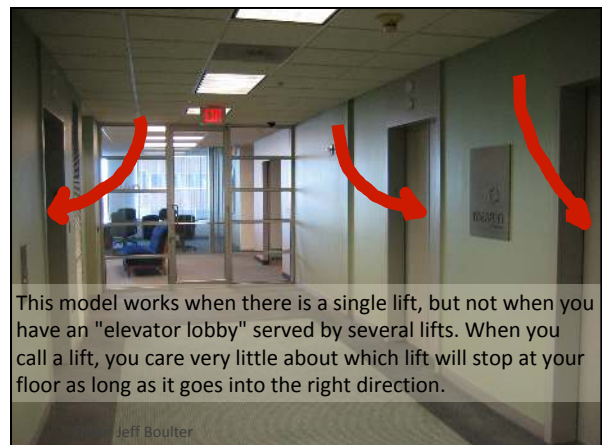
### Lift Class



**Floors served**  
**Direction**  
**Moving/Stopped**  
**Stop request**

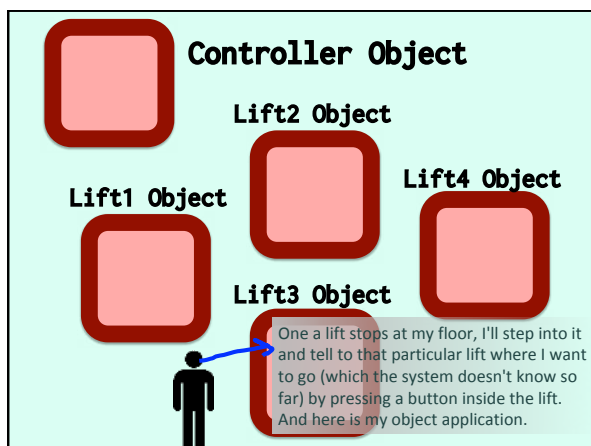
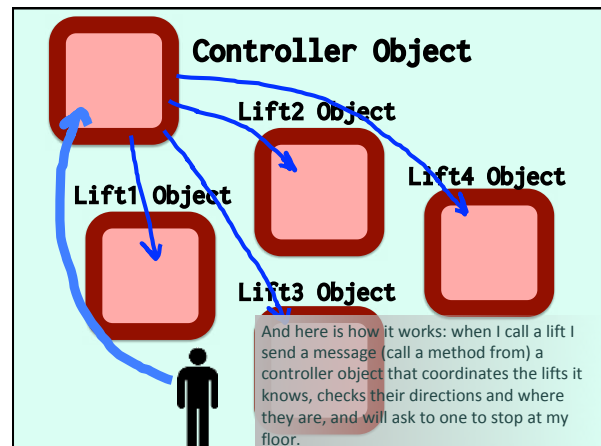
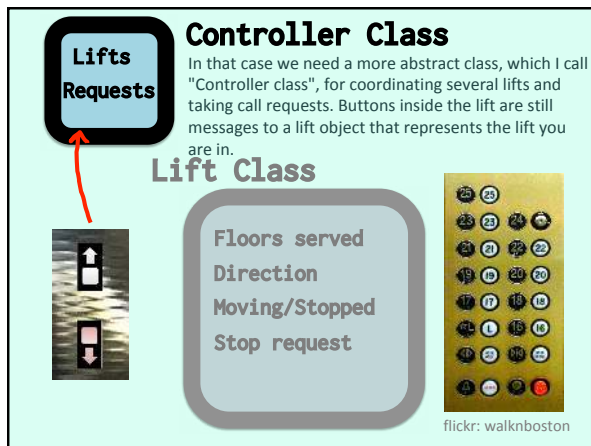


flickr: walknboston



This model works when there is a single lift, but not when you have an "elevator lobby" served by several lifts. When you call a lift, you care very little about which lift will stop at your floor as long as it goes into the right direction.

Jeff Boulter



The previous example has illustrated how I can start building a simple Object Oriented application, with concrete objects (lifts) and more abstract ones (controller) required for managing a pool of lifts. Then I have to set-up communications. I have said nothing of interactions between controller and lifts, but that's where the "intelligence" will be, and where we'll have to arbitrage between the-object-that-does-everything and chatty objects.

### Creating an Object Oriented Application

- Identify necessary objects
- Define their role (what they do)
- Define the data they need
- Set up communications

Maximize consistency and minimize coupling

↖ the hard part



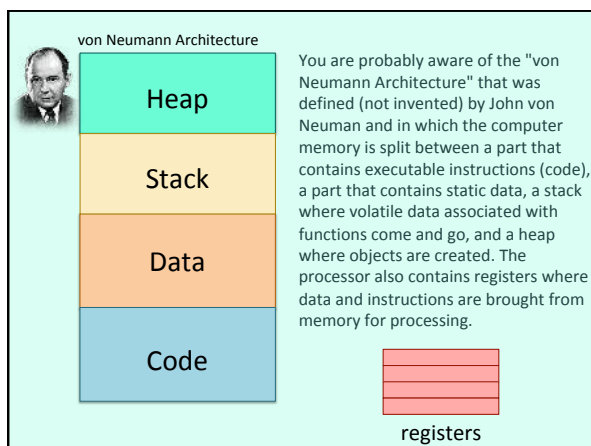
## Objects can model something real represent something abstract

It's important to understand that an object can correspond to a real-life entity ("Student Object", "Course Object") but that object-oriented programming can also be applied in more abstract contexts, such as the previous controller. Another fairly abstract class example is the "GregorianCalendar" class in the Java language, which represents not the kind of calendar that you tack to the wall but rules for date computations, which can be complicated. In programming, a network connection also becomes an object, and so forth.

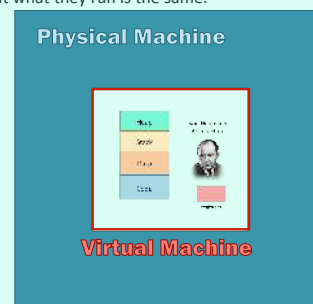
## The JVM in detail

(Textbook: Chapter 1, 1.13)

Before we move further to programming concepts proper, I'd like to review the architecture of the Java Virtual Machine (JVM) in some detail. It's important to understand how it works when you use some advanced functionalities.



The Java Virtual Machine simulates all this in a standard way on a lot of physical machines. Implementations of JVMs on different machines are different, but what they run is the same.



Hence what you sometimes hear about Java "Write once, run everywhere"

The JVM virtualizes a kind of ideal machine. There are other approaches to virtualization: simulating a physical machine in software (and installing any operating system on it) or simulating an operating system with "containers". Virtualization is an idea that can be traced back to the 1970s.

### Relatively "high-level" abstraction

Other approaches:

Simulating hardware in software

vmware

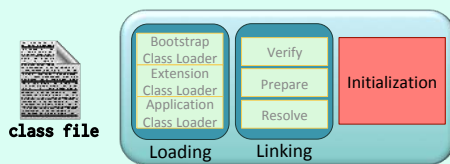
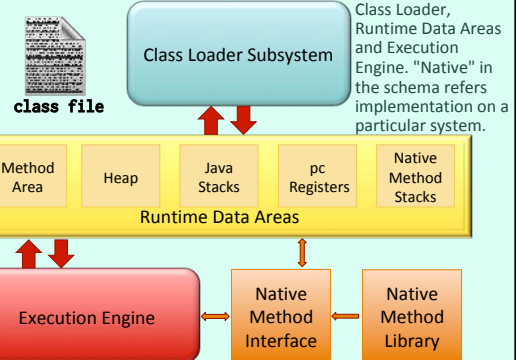


Simulating another Operating System



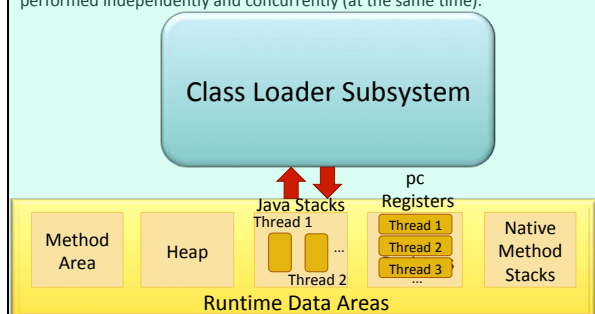
(containers)

## The Java Virtual Machine



The class loader is in charge of loading the .class files (sometimes grouped together into a .jar, **J**ava **A**rchive). Those files contain intermediate code which is no longer Java code but not yet computer instructions. One of the tasks of the class loader is to locate every single class referenced in the code (everything that was imported, but also the classes that are referenced but not defined in your program, and for which a .class is expected to be found in the CLASSPATH, a list of directories or .jar files), then to set up everything so that when you call a method the engine knows where to find the instructions to execute. Finally it will initialize everything and you'll be able to call main().

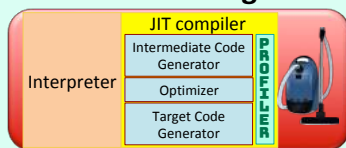
You find in the Runtime Data Areas all the components from the von Neumann architecture, with built-in support for threads, which are tasks performed independently and concurrently (at the same time).



You can have thousands of threads inside a single Java Virtual Machine.

The execution engine contains an interpreter that converts the "run everywhere" code found in the .class to some "run on this particular hardware" instructions that the processor can execute. Interpreting code is slow (especially when running loops and repeating instructions), and pretty soon was introduced in Java a "Just In Time Compiler" that converts but then reuses the conversion when the same code is executed again.

### Execution Engine



As the JIT compiler discovers the code as it executes, every optimization cannot be executed from the start, and there is a sophisticated mechanism. The last component is the garbage collector that identifies objects that are no longer in use and reclaims the memory they are using.

## Exceptions

(Textbook: Chapter 11)

After this description of the context in which Java programs execute, let's move to programming proper and exceptions. Exceptions are a way to change the normal path of execution of a program; they first appeared in the 1960s in a language called LISP (designed for artificial intelligence applications). Exceptions were introduced by Stroustrup in C++, and from there came into Java.

### Exception: **special object**

info about an error

kind of message passed back to calling method

An exception in Java is a special object, generated by an "exceptional event" (understand, most often, an error) that is passed back to the calling method through a mechanism different from the usual messaging between objects.

### Trigger an exception with **throw**



You throw an **object**,  
not a **type**

```
throw new MyException();
```

As an exception is an object, it must be instantiated (created from the template that the class defines) using **new** when you need it and **throw** it.

## Different Errors

### Compile-time

Syntax errors  
Wrong type

It's important to notice that in the large quantity of things that can go wrong in the development of a program, you have different types of errors. The javac program will tell you about wrong syntax, or incompatible types when assigning data or calling methods. You have to solve all these issues before you can run your program.

## Different Errors

### Compile-time

### Link-time

Before you can run your program, the Class Loader must load it and you may have passed compilation and failed linking, because the Class Loader fails to find a .class corresponding to objects you want to use.

## Different Errors

### Compile-time

### Link-time

### Run-time

### Logic

Exceptions

Detected by the application

Detected by the library

Detected by the Operating System / Hardware

When you run your program, you may run into other errors; one of your methods can detect that it gets parameters that are the right type but the wrong range of values. A built-in Java method may discover the same (and will throw an exception); or things may go really wrong and the operating system may discover it (hardware failure, for instance). Logic can also be wrong.

## Runtime ?

As some exceptions occur fairly often, some methods warn about it by using **throws** followed by the name(s) of the exceptions it can throw. The javac compiler is now aware of them.

Methods that throw exceptions tell it ...

```
class FileReader
    public FileReader(String fileName)
        throws FileNotFoundException
```

... so javac knows.

## Runtime ?

```
import java.io.File;
import java.io.FileReader;

public class IgnoredException {

    public static void main(String args[]) {
        FileReader fr = new FileReader("sample.txt");
    }
}
```

If you happily ignore the exceptions advertised by a method such as the previous constructor ...

## Runtime ?

```
$ javac IgnoredException.java
IgnoredException.java:7: error: unreported exception
FileNotFoundException; must be caught or declared to be
thrown
    FileReader fr = new FileReader("sample.txt");
                        ^
1 error
$
```

... javac will simply not let you do it. The message is pretty explicit: it wants you to either deal with the problem, or let the world know that something may fail and that another object has to deal with it.

So we have two options



You must either include the method call that can fail into a **try ... catch** block and deal with the exception in the **catch** block.

Deal with it (**try ... catch ...**)

So we have two options



Or you may get away with handling the exception if you warn your callers that what you are doing for them may fail, and that in that case *they* will have to deal with the problem.

Warn callers (**throws ...**)

## CHECKED EXCEPTION

handled  
OR  
declared as thrown

When you are faced with this choice you are dealing with what is known as a checked exception, and javac will make sure that you follow the rules.

but something else can happen

Compile-time

Link-time

Run-time

Logic

Detected by the application

Detected by the library

Detected by the Operating System / Hardware

Exceptions

Errors

**java.io.IOException**

When you read a file, it may exist but be corrupted.

## NEXT TIME

\* A few more things to say about exceptions (book Chapter 11)

\* I'll talk about javadoc, a tool for easily generating a good documentation for your programs.

\* Recursion (book Chapter 18) – those who have already taken C/C++ with me will retrieve (with pleasure, no doubt) some familiar material.

\* If time permits, we'll start with Generics (book Chapter 21), which are the Java equivalent (not quite, but superficially similar) of templates in C++.

Whether you want to take a look at the book before or after the lecture is entirely up to you.