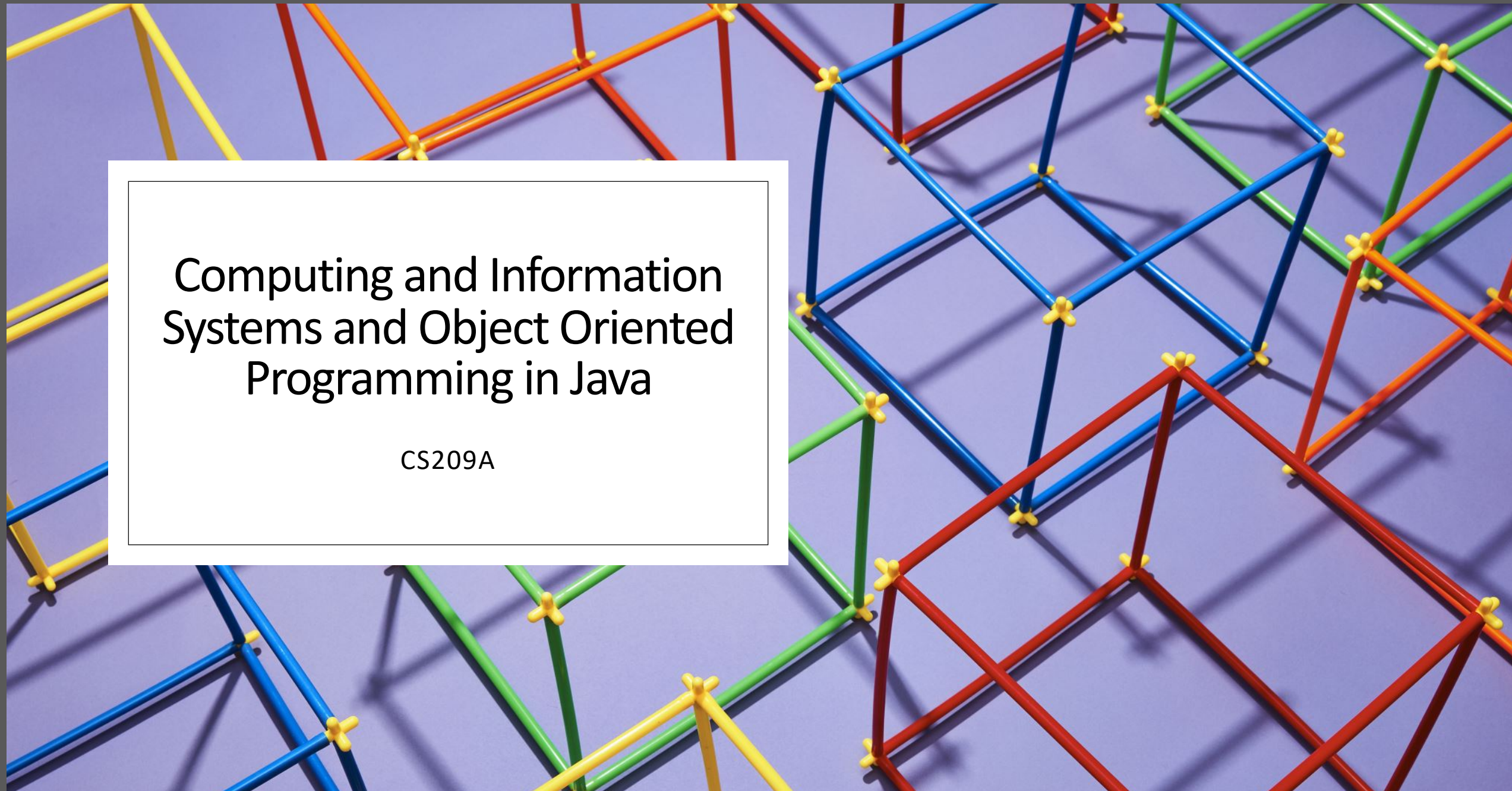


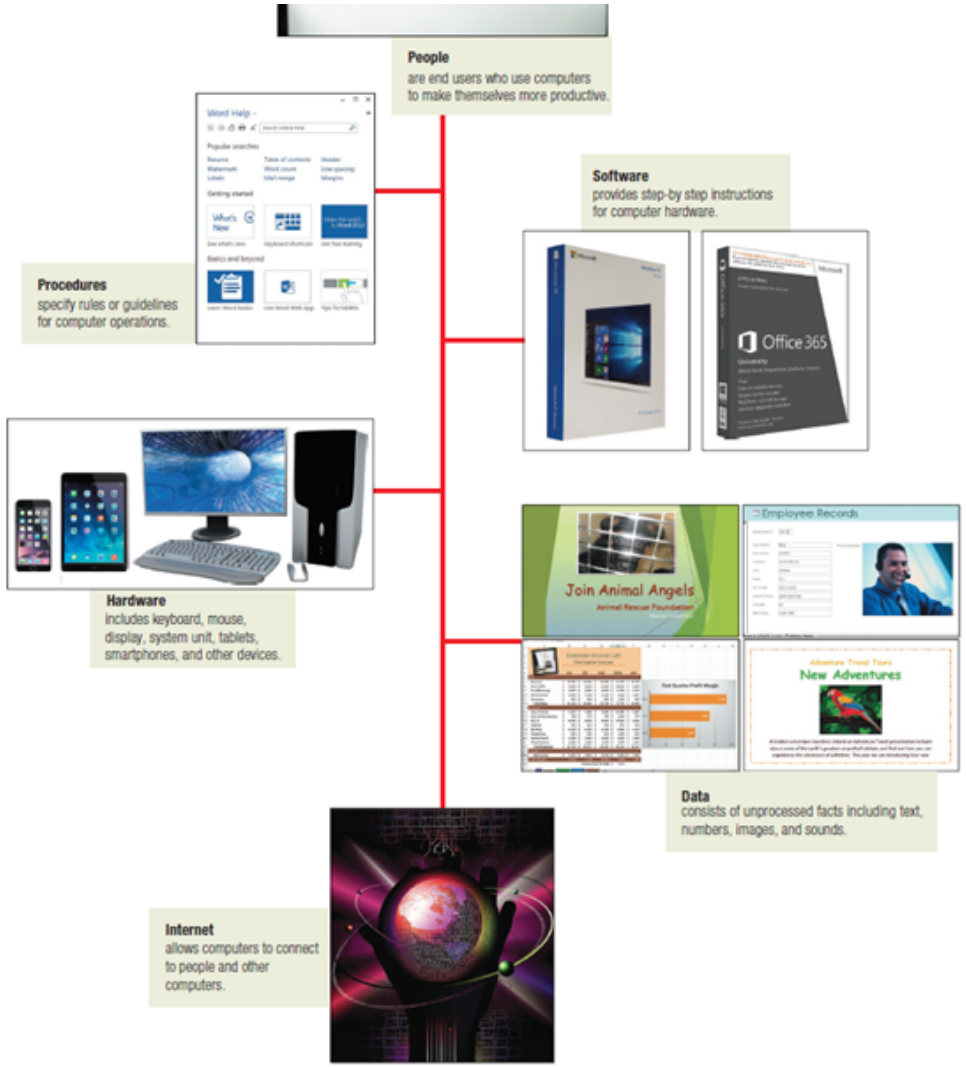
Computing and Information Systems and Object Oriented Programming in Java

CS209A

Information Systems

- Computing takes place in information systems
- This presentation provides a brief overview of this context

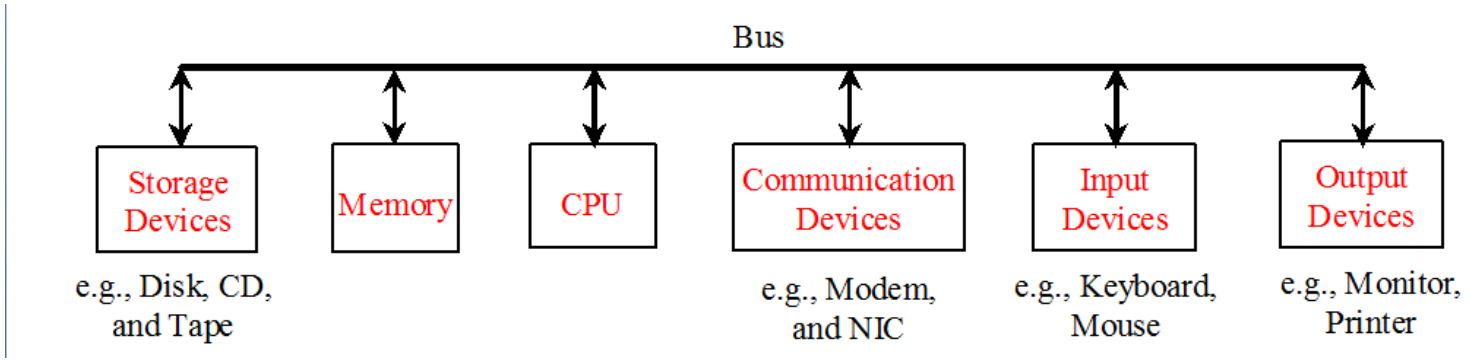
- Notes and reading:
  - Text B Chapter 1



An Information System

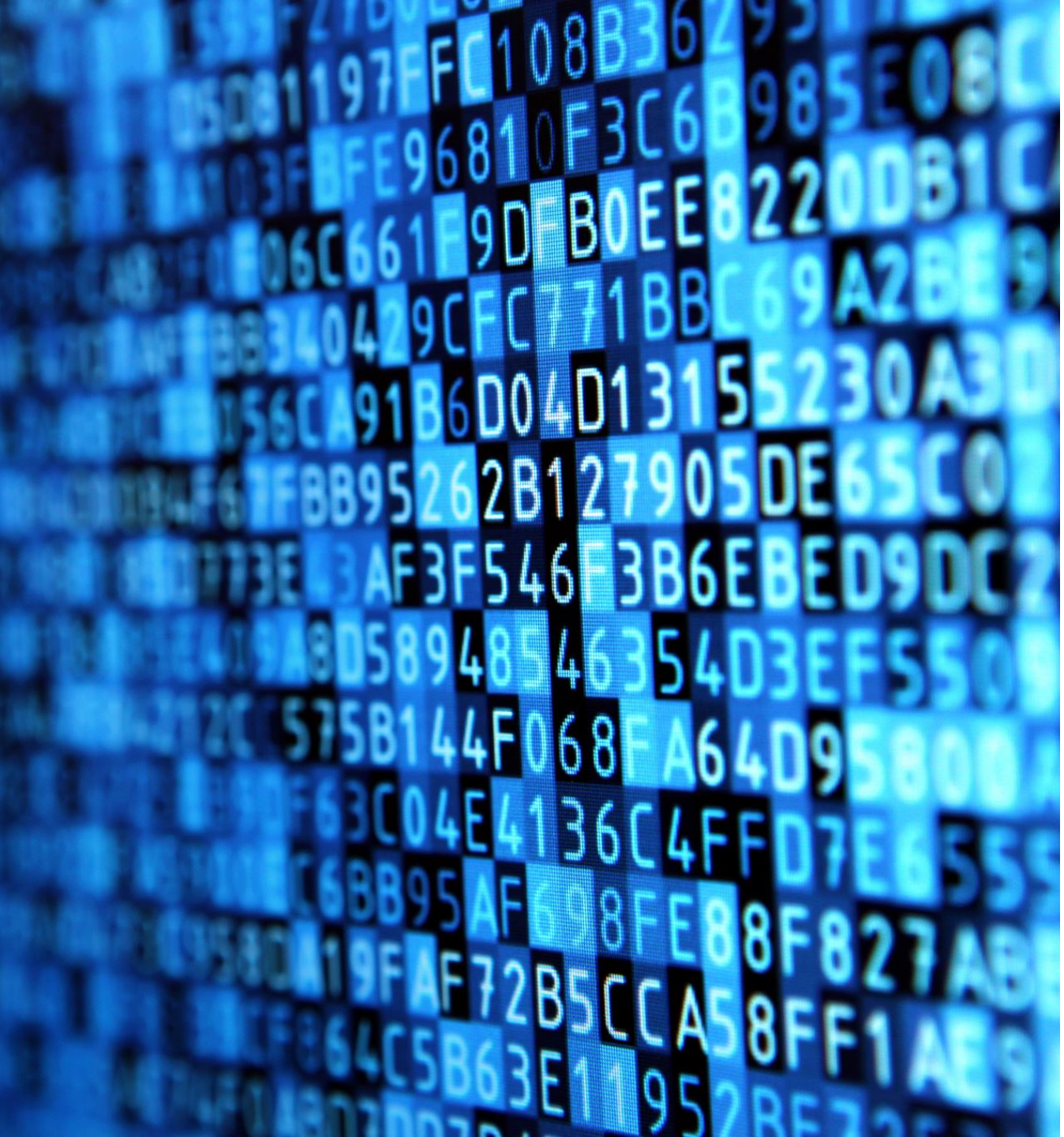
- People
- Procedures
- Software
- Hardware
- Data
- Networks

- Data VS Information



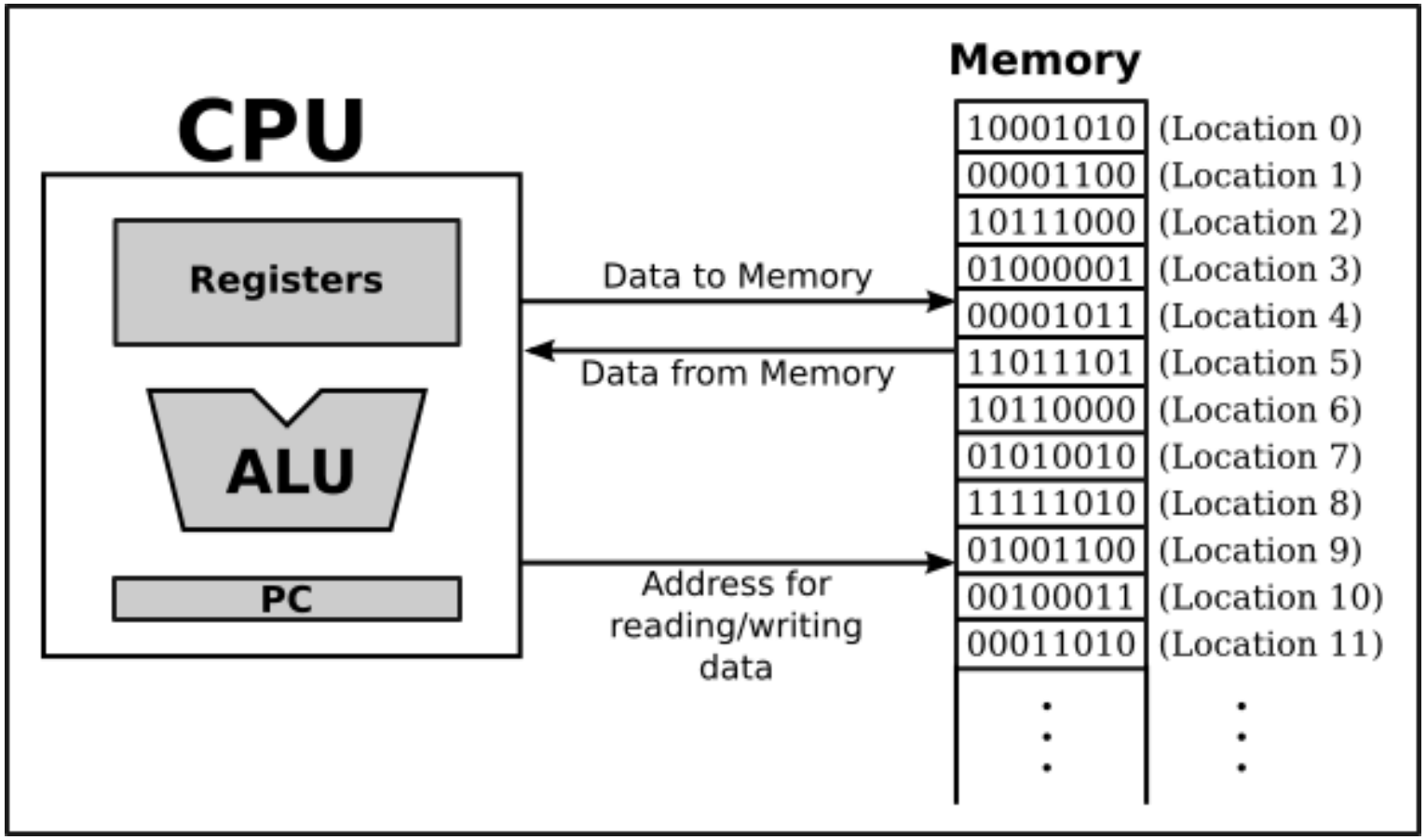
Hardware

- A computer consists of various devices referred to as hardware
  - e.g., the keyboard, screen, mouse, disks, memory, DVD, CD-ROM and central processing units (CPU)



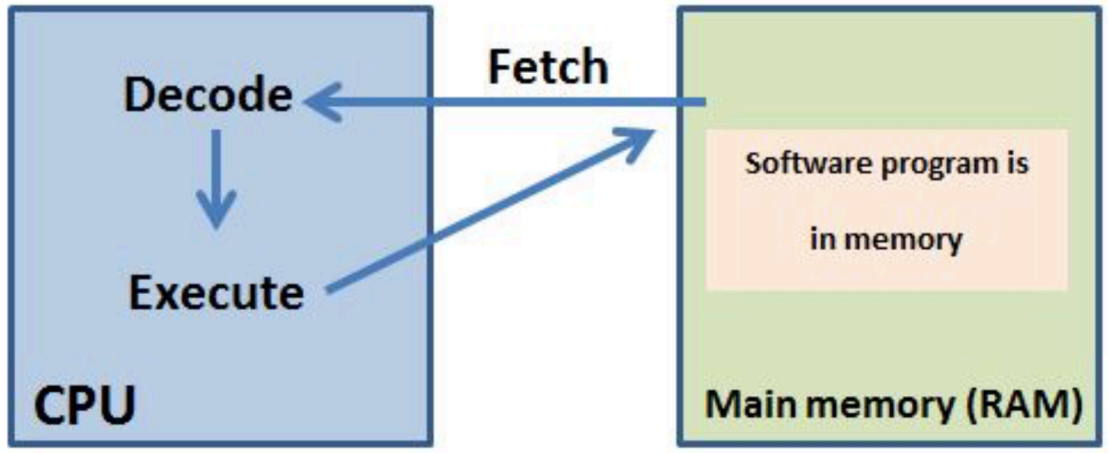
Software

- Programs that run on a computer are referred to as software
- A program is a sequence of instructions that specify how to perform a computation
- Instructions can include statements to:
  - Input – get data
  - Output – output data
  - Math – perform calculations
  - Testing – check for conditions or run statements
  - Repetition – perform repetitive actions



Model for how computers work

- Main memory holds machine language programs and data encoded as binary numbers
- The CPU fetches instructions from memory in sequence and executes them
- Each instruction is a very small task (e.g. adding 2 numbers)
- The program must be complete and unambiguous as the CPU executes everything exactly as written



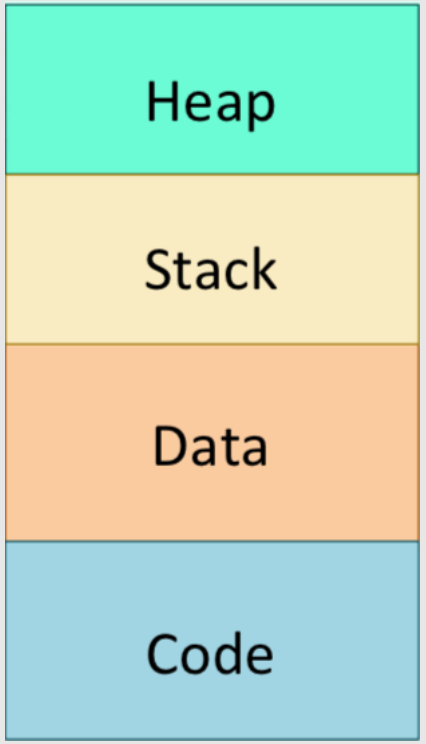
Fetch Execute Cycle

- **The Fetch Execute Cycle** is the basic cycle of how computers operate to process instructions
- During the cycle the computer receives a program instruction from memory, then it establishes and carries out the actions specified
- **Fetch** – gets the next program command from the computer's memory
- **Decode** – deciphers what the program is telling the computer to do
- **Execute** – carries out the requested action
- **Store** – saves the results to a Register or Memory

John von Neumann

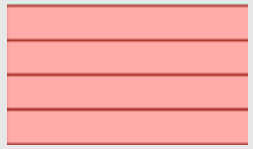
- The **fetch execute cycle** was first proposed by John von Neumann who is famous for the Von Neumann architecture, the framework which is being followed by most computers today.

Von Neumann Architecture



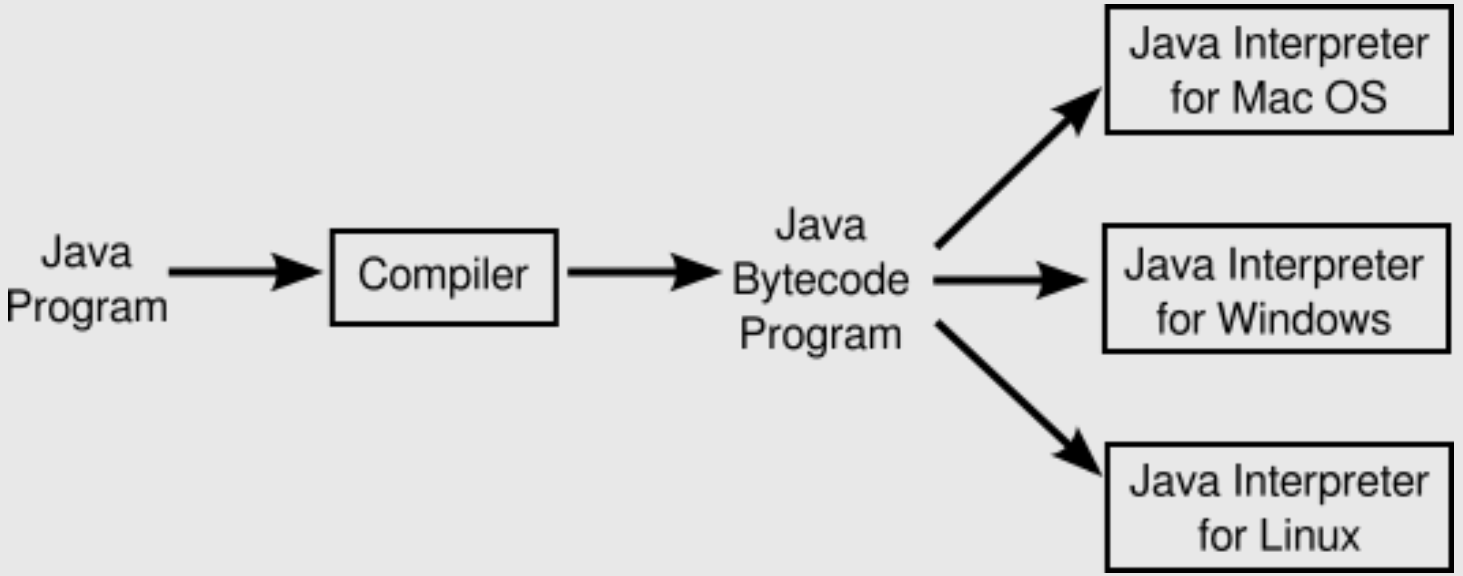
You are probably aware of the "von Neumann Architecture" that was defined (not invented) by John von Neuman and in which the computer memory is split between a part that contains executable instructions (code), a part that contains static data, a stack where volatile data associated with functions come and go, and a heap where objects are created. The processor also contains registers where data and instructions are brought from memory for processing.

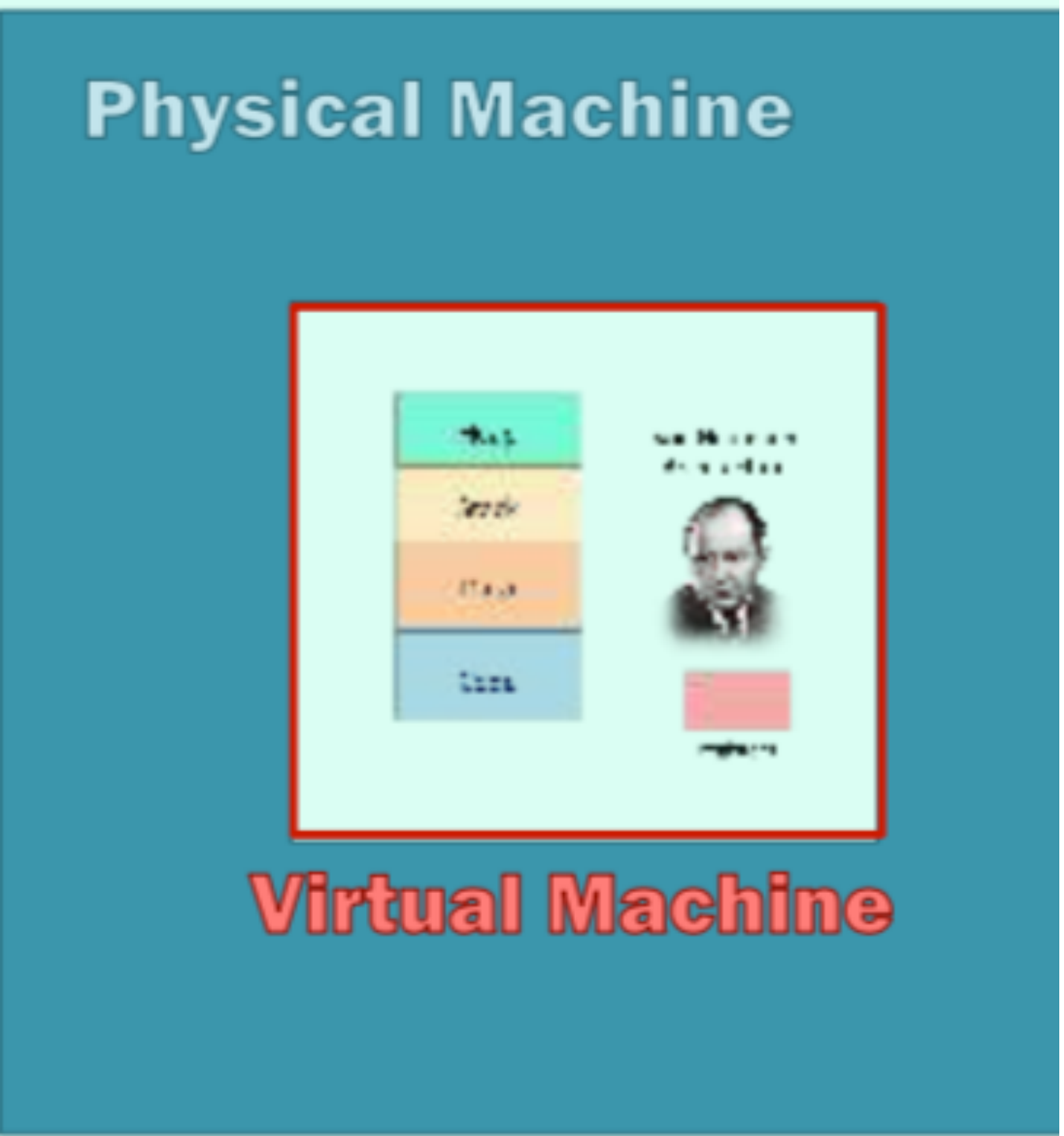
Registers



Java Virtual Machine

- The CPU executes instructions written in Machine Code (a very simple set of instructions that can be executed directly by the CPU)
- Usually programs are written in high-level programming languages such as Java, Python, or C++. A program written in a high-level language cannot be run directly on any computer and has to be translated.
- This translation can be done by a program called a compiler.
- Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language).
- Instead of using a compiler, we can use an **interpreter**, which translates each instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.





JAVA Virtual Machine

- The Java Virtual Machine simulates a Von Neuman machine 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.
- It is often said “With java – write once – run everywhere”

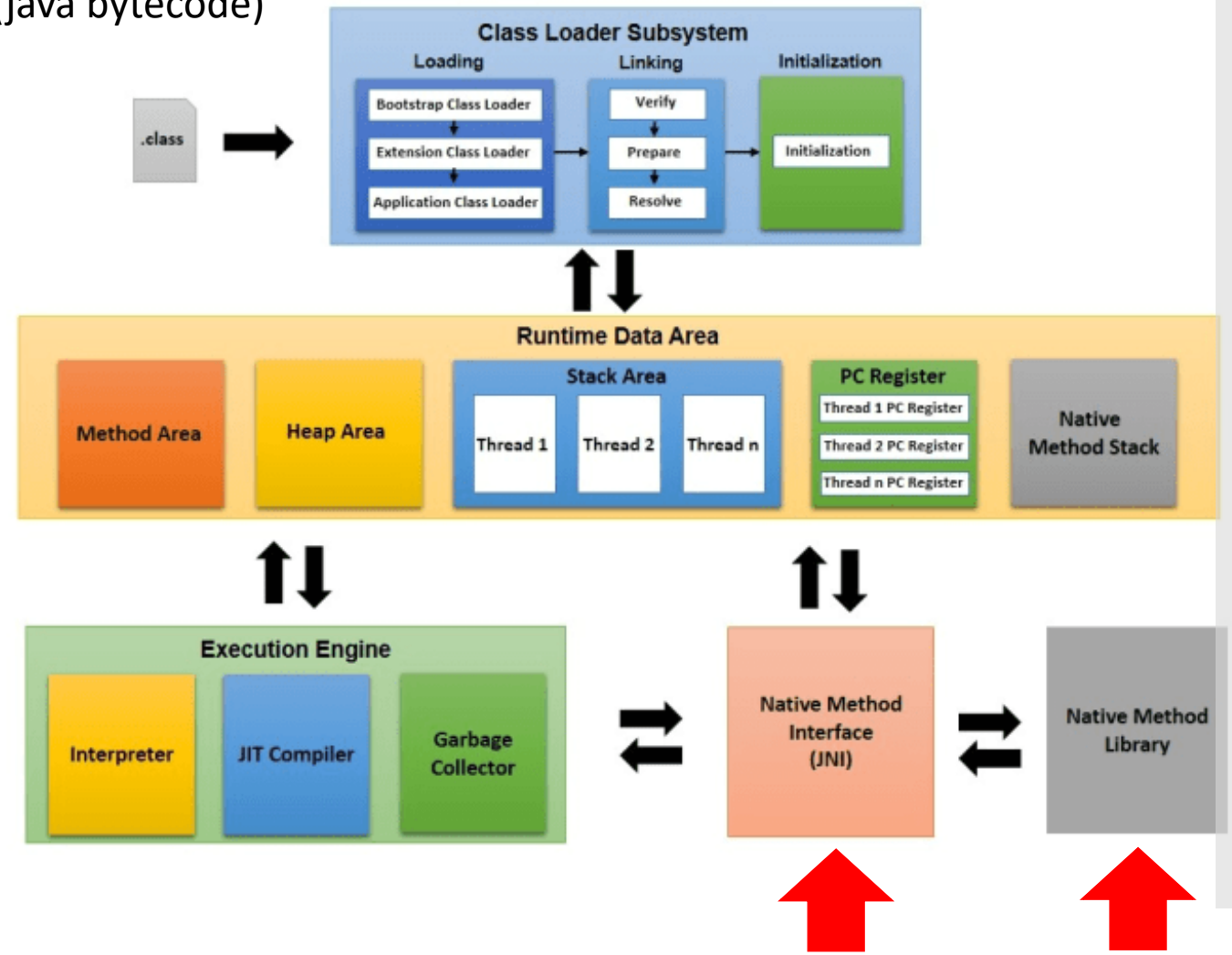
Other Examples of Virtualization

- In essence the JVM is a virtual representation of a type of ideal machine
- Virtualization hides the physical characteristics of a computing platform from the users, presenting instead an abstract computing platform
- There are other approaches to virtualization such as:
  - Simulating a hardware system in software and installing any operating system on it (eg Vmware)
  - Simulating another different operating system on machine (eg Docker and contntainers)

How the Java Virtual Machine Works

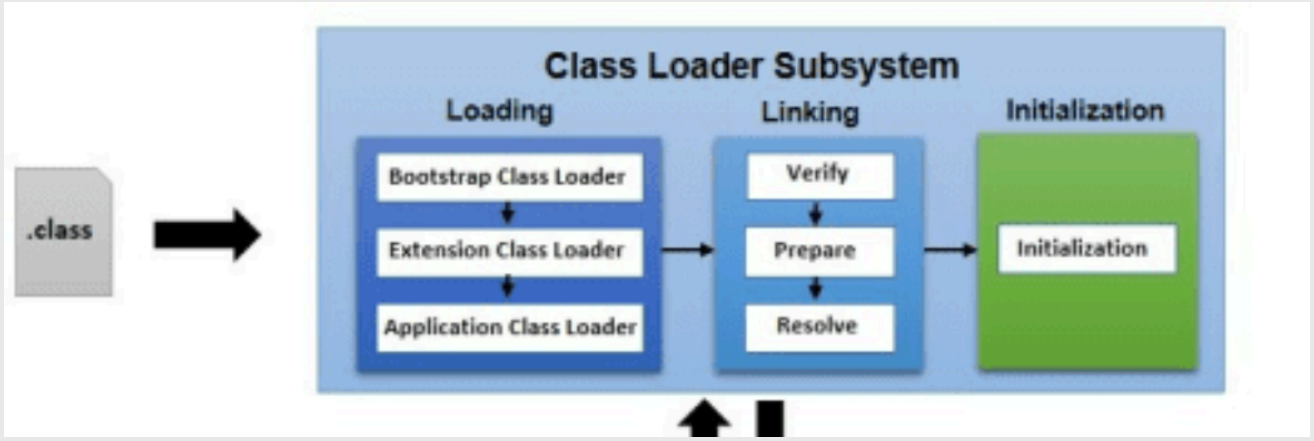
A JVM can be divided into three main components, Class Loader, Runtime Data Areas and Execution Engine. "Native" in the schema refers implementation on a particular system.

Class file
(java bytecode)



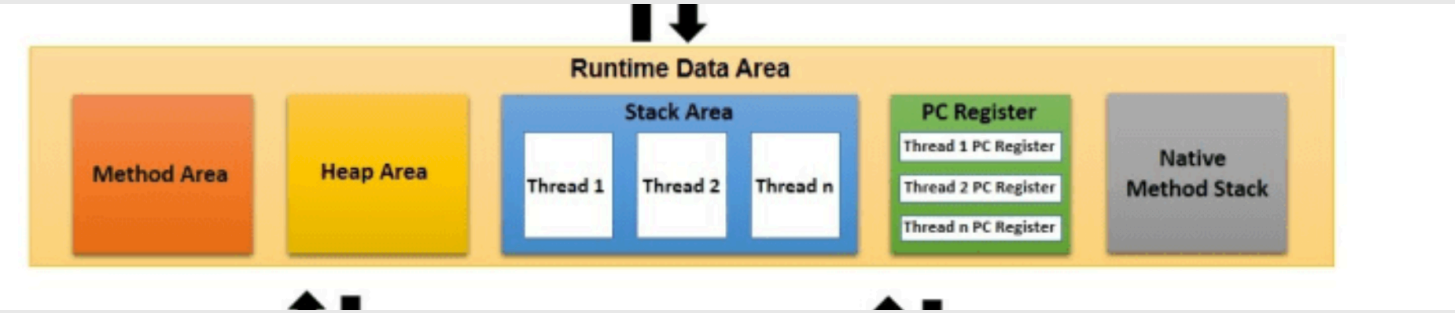
Class Loader Subsystem

- The class loader is in charge of loading the .class files (sometimes grouped together into a .jar**, J**ava **AR**chive). 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().



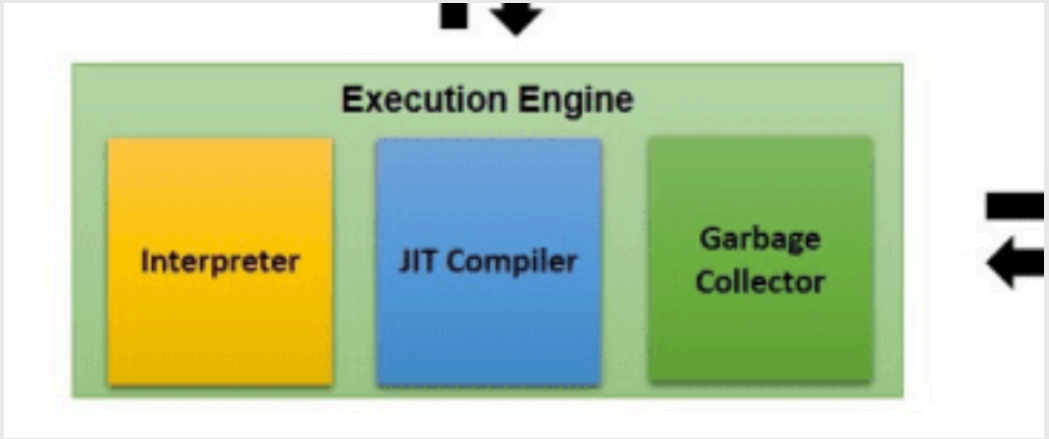
Runtime Data Areas

- You will 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).
- A single machine can have many thousands of threads!



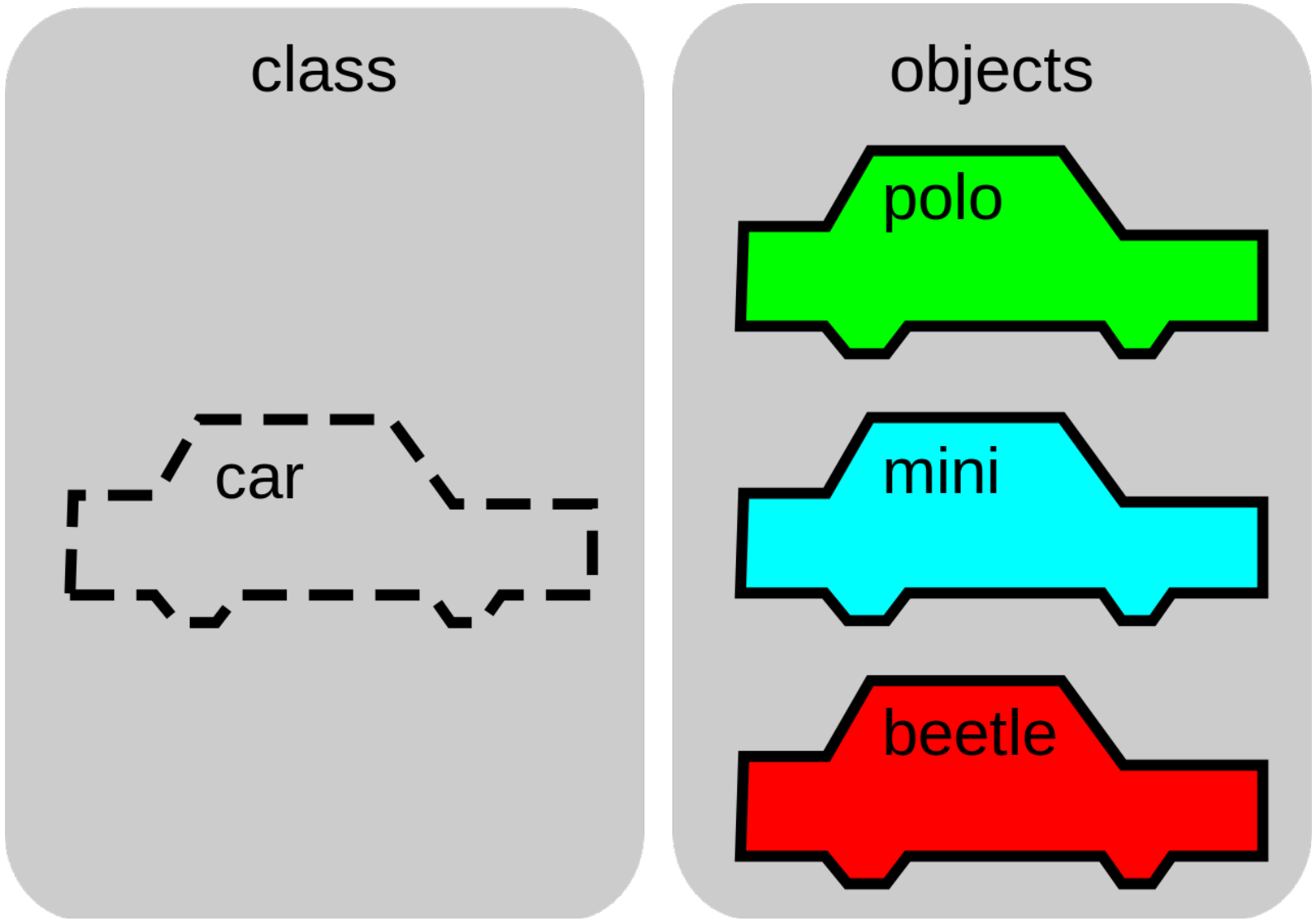
Execution Engine

- The execution engines contains an interpreter that converts the "run everywhere" code found in the .class to some "run on this particular harddware" 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.
- 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.



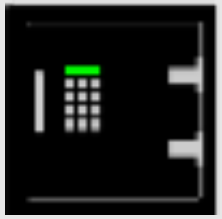
Object Oriented Programming

- Java is especially suited to object oriented programming paradigms
- "Object Orientation" can be contrasted with procedural (or functional) orientation
- 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.


class
car
objects
polo
mini
beetle

Objects

Identity



Object Data

Provides services

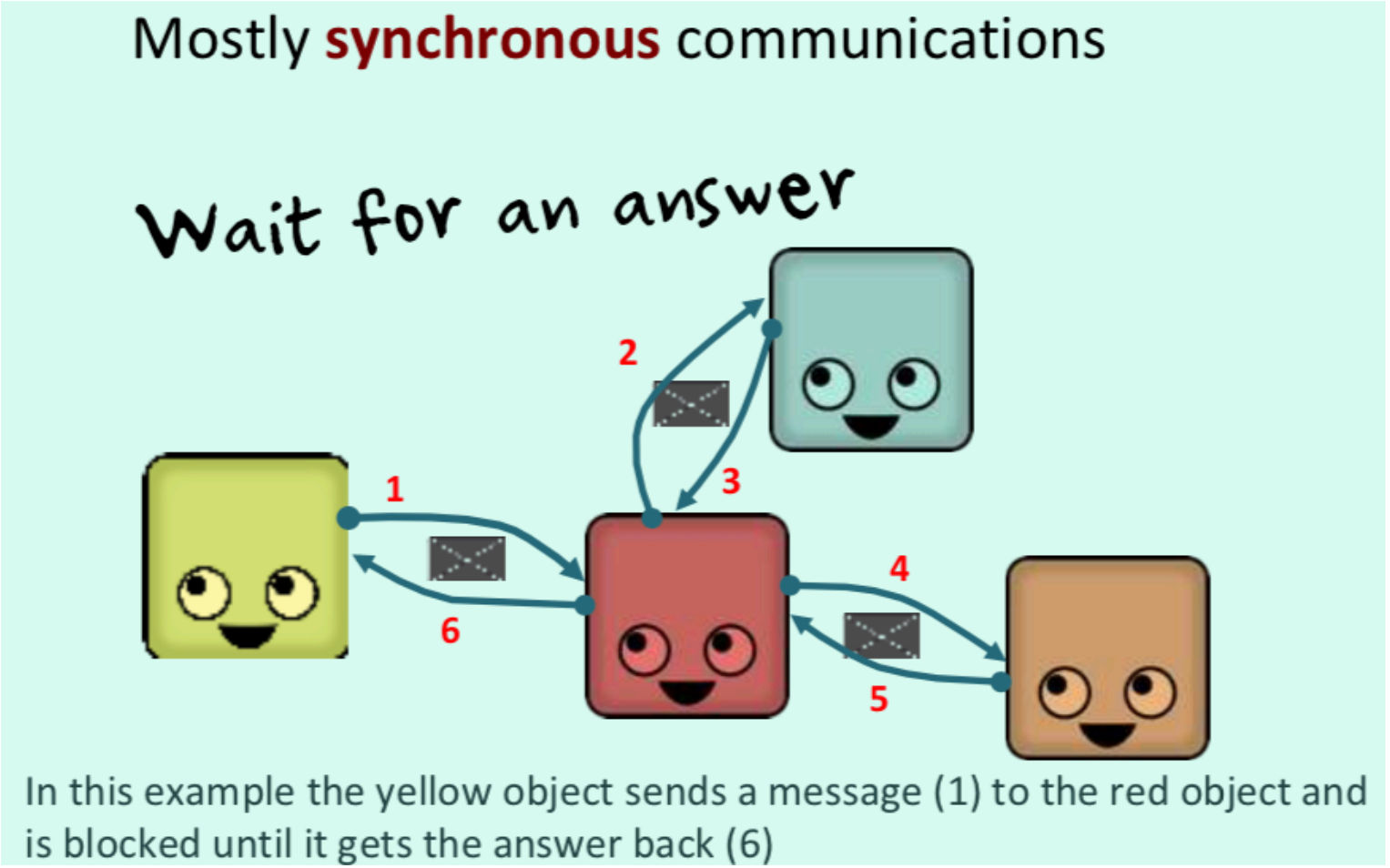
Message

- In Computer Science, an Object is a software component that has an identity, stores data that is usually carefully hidden ("private") and provide services. The object provides services through messages, which are the method calls. Of course, to send a message you must know whom to send it, hence the identity, called "reference" in Java.

- An object can provide services to itself and in this case the identity is “this”, which means “me, this object”

Messaging

When messages are exchanged, they are **synchronous** (a word that comes from Greek and means "same time"), which practically means that the caller waits for the answer and does nothing until it gets it.

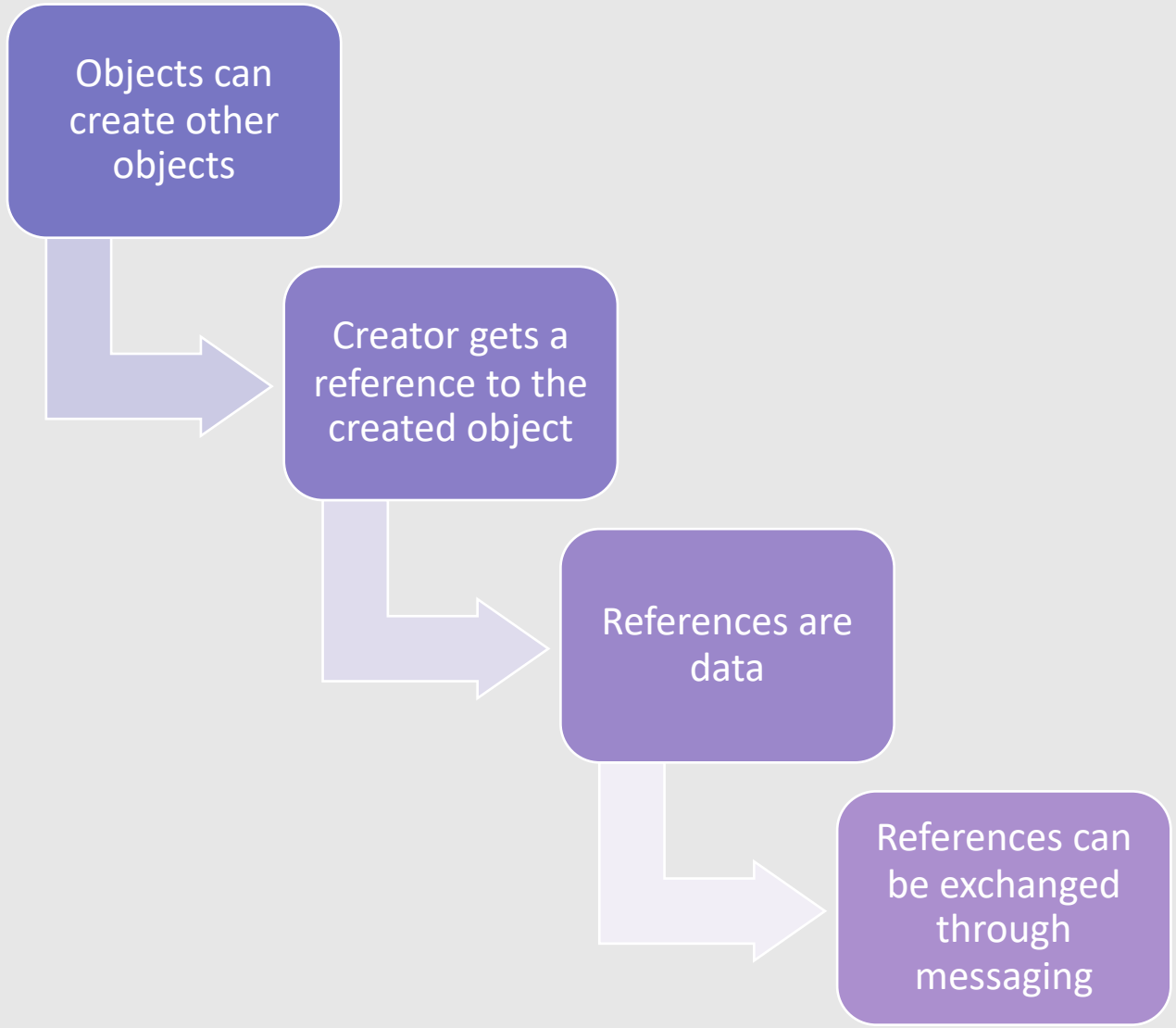


Coupling

- When many objects exchange messages between each other it's said that there is a higher degree "coupling" and it can be very slow (especially when, as it happens sometimes, the objects aren't all running in the same machine).
- On one hand it can be better to divide a big task in many elementary tasks executed by different objects, but on the other it increases the number of messages to exchange.
- Higher coupling = Slower programs

Object References

- 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".



Public static void main(String args[])

- 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.
- The special object (*main*) creates the first objects.
- A **bootstrap** is the program that initializes the operating system (OS) during startup.
- The term **bootstrap** or **bootstrapping** originated in the early 1950s. It referred to a **bootstrap** load button that was used to initiate a hardwired **bootstrap** program, or smaller program that executed a larger program such as the OS.

Test.java

```
1  package com.journaldev.java.examples1;
2
3  public class Test {
4
5      public static void main(String args[]) {
6
7          System.out.println("Hello World");
8
9      }
10 }
```

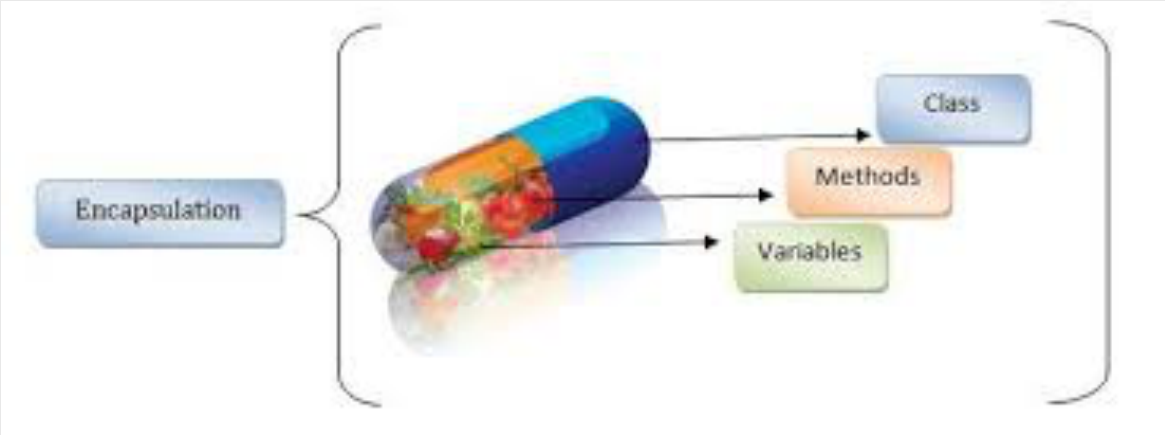
Designing OO Programs

- 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 PRINCIPLE: Encapsulation

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



Encapsulation

Car

model

speed

engine

speedLimit

drive()

stop()

setSpeed(number)

Object Oriented Programming…

Entities

Having state

Having a behaviour

Exchanging messages

Challenges

- 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.

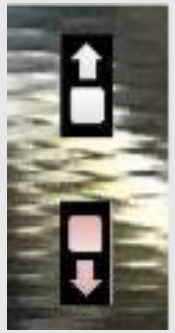
Lift Example

Problem Description

If you work for a lift manufacturer, you might 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. One complicating factor is the need to model peak usage as people tend to use the lifts at the same time. Modelling and simulation can be used to do this, which takes us back to the roots of Object-Oriented programming.

Implementation

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.



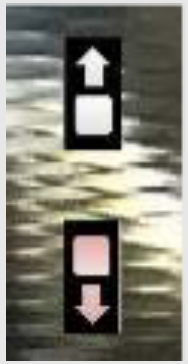
Lift Class

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

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.

Controller Class

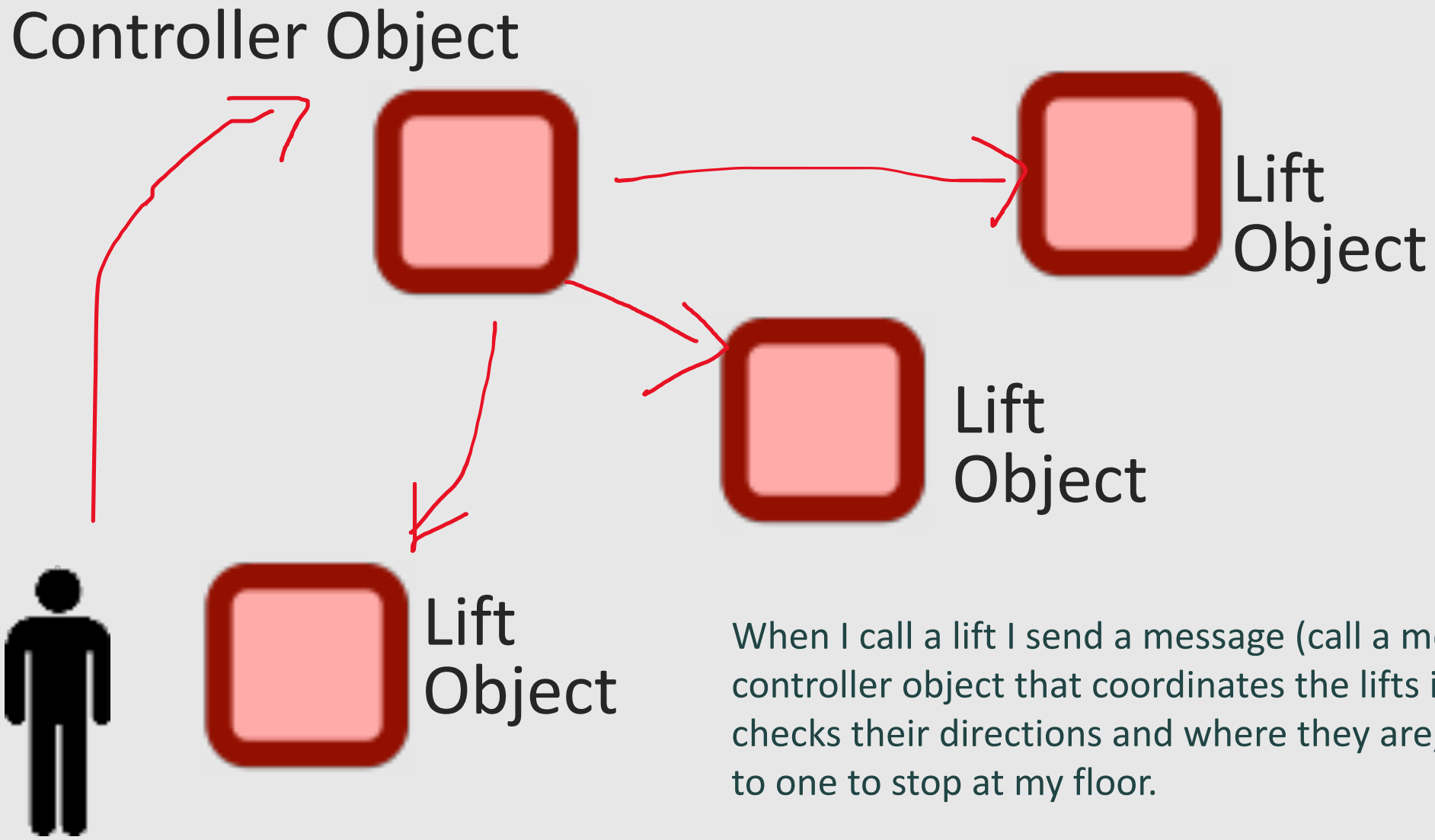
- Lift Requests



In that case we need a more abstract class, which I call "Controller class", for coordinating several lifts and taking call requests. Buttons inside the lift are still messages to a lift object that represents the lift you are in.

Lift Class

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

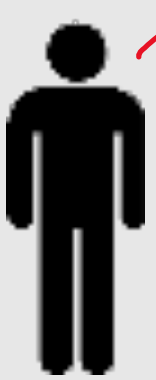


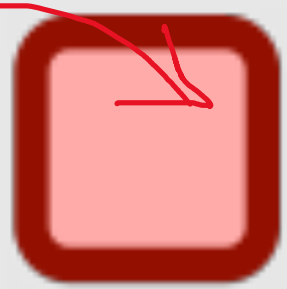
When I call a lift I send a message (call a method from) a controller object that coordinates the lifts it knows, checks their directions and where they are, and will ask to one to stop at my floor.

Controller Object

Lift
Object

Lift
Object





Lift
Object

When a lift stops at my floor, I'll step into it and tell to that particular lift where I want to go (which the system doesn't know so far) by pressing a button inside the lift. And here is my object application.

Creating an Object Oriented Application

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

the hard part

- **Maximize consistency and minimize coupling**
- Trade-off between one object that does everything and objects that send too many messages