Lecture 14

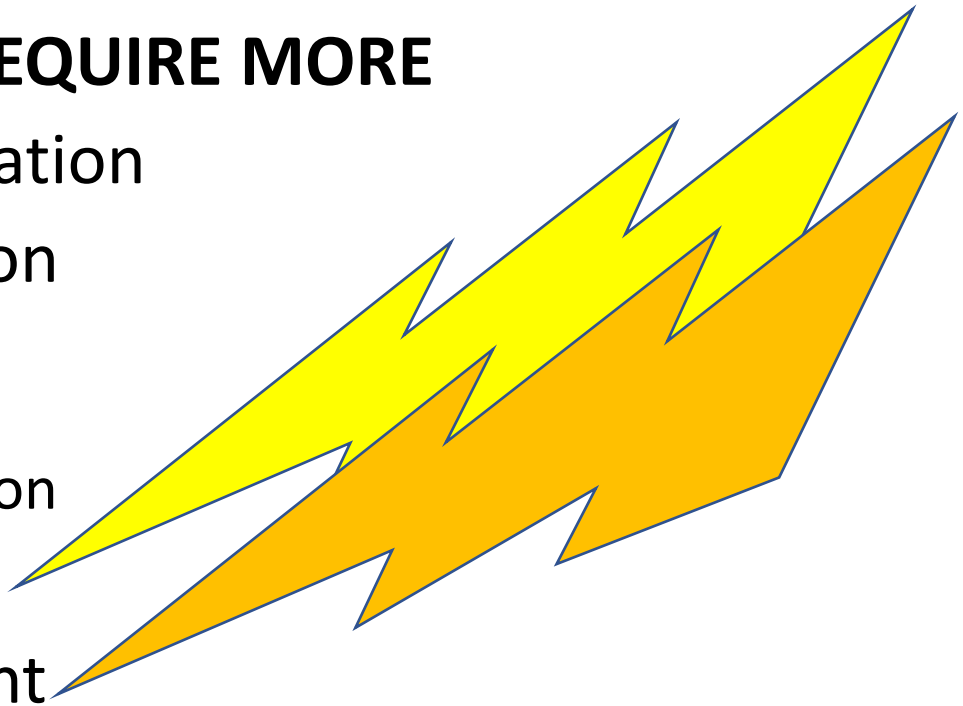
Regex

Review

Writing code is one thing. But…

PROJECTS REQUIRE MORE

- Documentation
- Organization
  - Code
  - Build
  - Integration
- Testing
- Deployment



In general, and with any language, a project is more than writing one program and there is a whole eco-system around it.

Testing

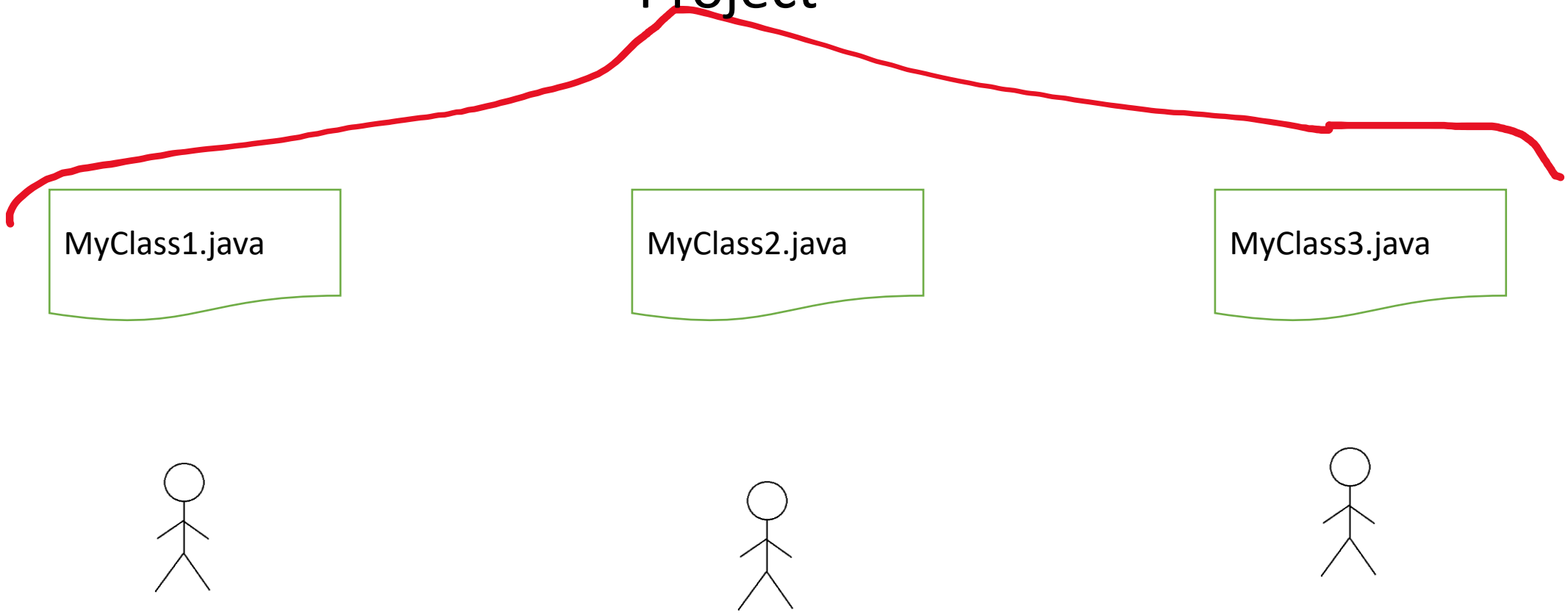
Testing

A very important phase in the life of a developer is testing, which is both kind of boring and difficult to do properly.

Testing is a task that has to be done repeatedly, because very often a change (new feature, bug fix) breaks something that used to work.

We can use "test suites" that just run the software through a lot of controls (essentially automated tests) and checks that everyone of them is passed.

Integration Testing



Testing when several developers are working on the same project.
The goal is ensuring that all the parts will work together.

Integration Testing

- Defects in the interfaces / interactions between components or systems
- Components and modules developed internally
- Interfaces with external products or data services etc

The fastest developer cannot wait on the slowest one to test the code!

In general its important to test the code as early as possible, even if you are using objects and methods currently being developed by someone else

(remember that object-oriented programming is mostly objects exchanging messages by calling methods)

Mock (or dummy) classes and methods for testing

The solution is to write very simple objects and methods just for simulating the real thing (that is not ready or in another component). EG instead of actually getting data from a DB you can return the same data from a small hard-coded or configured collection. Instead of getting a real message from a remote server you can make a fake one. The word ”mock” meaning imitation or fake is often used, it can be found in the names of products that are useful for generating (with reflection) code that is used in testing.

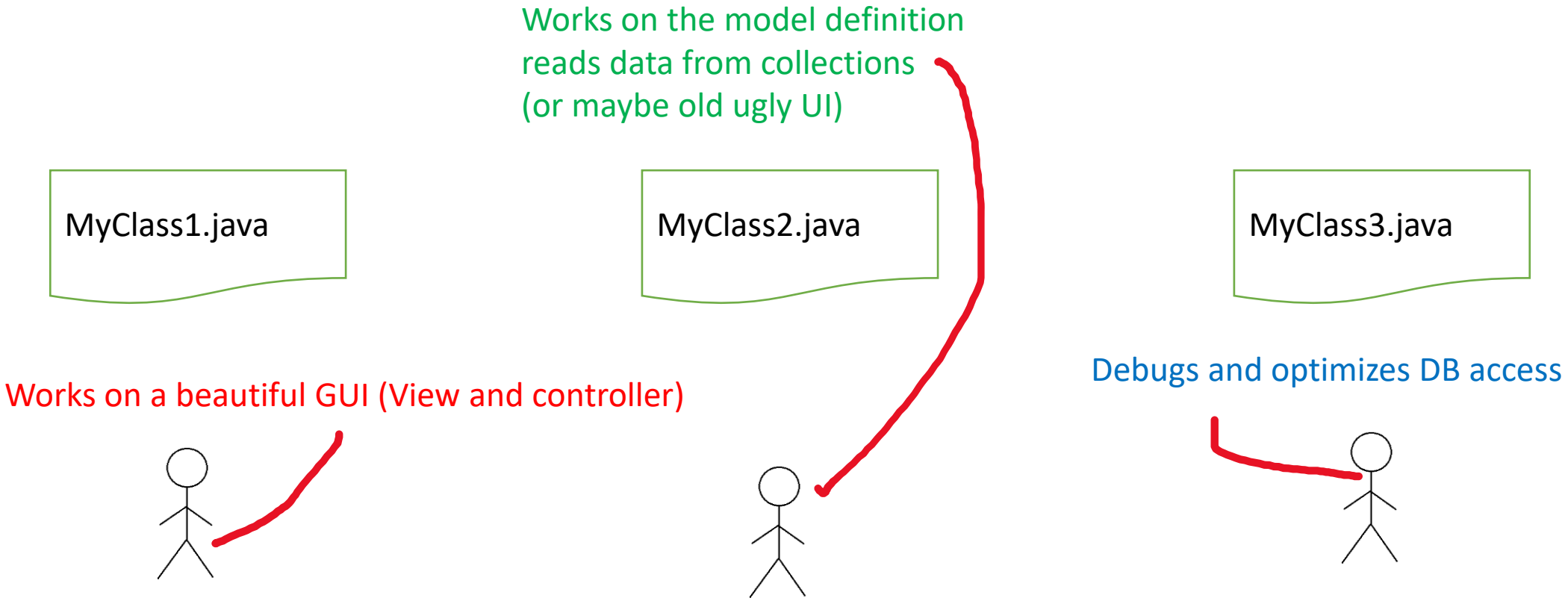
Mock的使用背景

单元测试的思路就是我们想在不涉及依赖关系的情况下测试代码。

Mock测试就是在测试过程中，对那些不容易构建的对象，用一个虚拟对象来代替测试的情形。

Unit testing and mocking

Testing



This allows developers to test their code without having to wait for others – or without having written all their methods.

IT is necessary there is a design that specifies how the components interact

Code design for testing: **Dependency Injection**

An important idea for testing is that when an object depends on another object from a different class, it should not create the other object, but should get a reference to it. The dependency is "injected" (passed).

This makes testing far easier, because you don't have to worry about what the constructor should look like and what arguments it takes. Dependency injection is central to some development frameworks such as Spring and is considered a good development practice.

Transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.

Aspects of Testing

Testing covers many fields – including the behaviour when something that wasn't expected happens.

- Expected outcome/ result?
- New change doesn’t break something? (regression)
  - Potentially a very large set of small tests that are run to make sure when a new piece of code is added to a large system it does not cause it to regress to be less useful (that is to reduce the current functions it is expected to have)
- What the user wanted?
  - User Acceptance Test
- Correct performance
  - EG load testing

Some tests are usually carried out by support teams, not by developers themselves. For the testing part that directly regards developers, some tools exist that are based on annotations.

- Types of testing:
  - Unit testing
  - Integration testing
  - System testing
  - Stress testing
  - Acceptance testing (should be based on requirements)
  - …
  - Smoke testing
  - Performance testing
  - Regression testing

Smoke testing

There are many different testing terminologies used in the software industry…

Smoke testing, sanity testing, etc etc

Smoke testing, the **preliminary level of testing is done to ensure whether the major functionalities of a build/software is functioning properly**. In simpler words, we can say that it is an entry criteria for ensuring that a software is ready for further testing. Confidence testing, Build Verification Testing are other names for smoke testing.

Smoke Testing 在软件测试中的意义，应该说取的是其原始概念中的目的而非手段。通过 Smoke Testing，在软件代码正式编译并交付测试之前，先尽量消除其“表面的”错误，减少后期测试的负担。因此可以说，Smoke Testing 是预测试。

https://www.cnblogs.com/zzp28/articles/1742661.html

Testing Tools

JUNit is widely used and you should be familiar with it

There are of course other testing platforms, e.g. TestNG

JUnit

```
import static org.junit.Assert.*;
import org.junit.*
```

class MyClass

```
public int method1{
}

public int method2{
}
```

```
@test
public int method1{
}

@test
public int method2{
}
```

The idea is to mirror a class with a test class that checks, in a test method, a method from the original class.

Test Methods

A test method, annotated as such, uses an assert*xxx*() function to compare the result of a method to test to an expected result.

```
@Test
public void testmethod1 {
        // Create object,
        // initialize parameters ...
        assertEquals(expected_result, obj.method1(...));
}
```

There are many assert*xxx* methods, that can optionally take a messsage as parameter.

- assertEquals("message",A,B);
- assertTrue(A);
- assertFalse(A);
- assertNotNull(A);
- ...

Junit Annotations

@Test

@Before

@After

@Test(expected = Exception.class)

@Test(timeout = 100)

Annotations allow to define "before" and "after" operations, and even to check that we are getting the proper exception.

Testing Setup

- **"Test suites"**
- **Ideally test only one class**
- **As many test methods as you need**

Normally you are supposed to test one class at once, and you can have multiple test methods to test different aspects (there is NOT a one-to-one correspondance between methods being tested and test methods).

Tests can be run from multiple environments.

- Running JUnit Tests (Tests can be run from multiple environments).
  - IDE
  - Build tool
  - Command line: $ java org.junit.runner.JUnitCore TestClass1 [...other test classes...]

  - You can use Maven to automatically run a test suite when you build your application (this is a common practice)

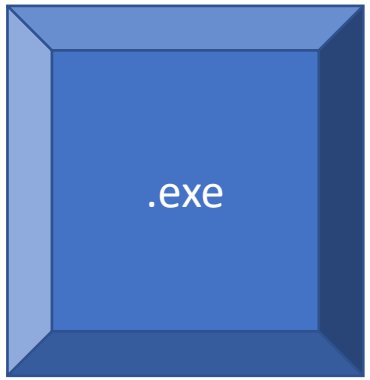
Deployment

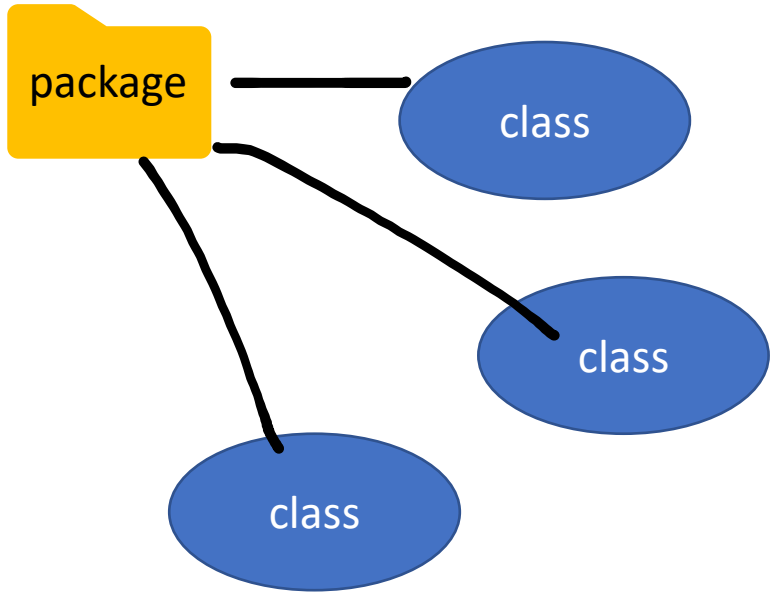
Deployment: *Distributing the program*

The last aspect isn't the least important. How are you going to distribute your program? Cases when the program is a single .class are very rare. Usually you need quite a number of files to successfully run a program.

Deployment: Executable Files

Compared to a standard .exe file in Windows, Java is a mess. You may need several .class files, as well as one or several packages, to successfully run your program.





Deployment: Jar Files

- When you need to send files by email you sometimes zip them into an archive.
- You do the same thing in java with a .jar file
- The JVM knows how to read and execute a .jar file without having to unzip it first.

- Either a complementary library (e.g. JDBC drivers)

  java –cp somefile.jar

- Or the main program

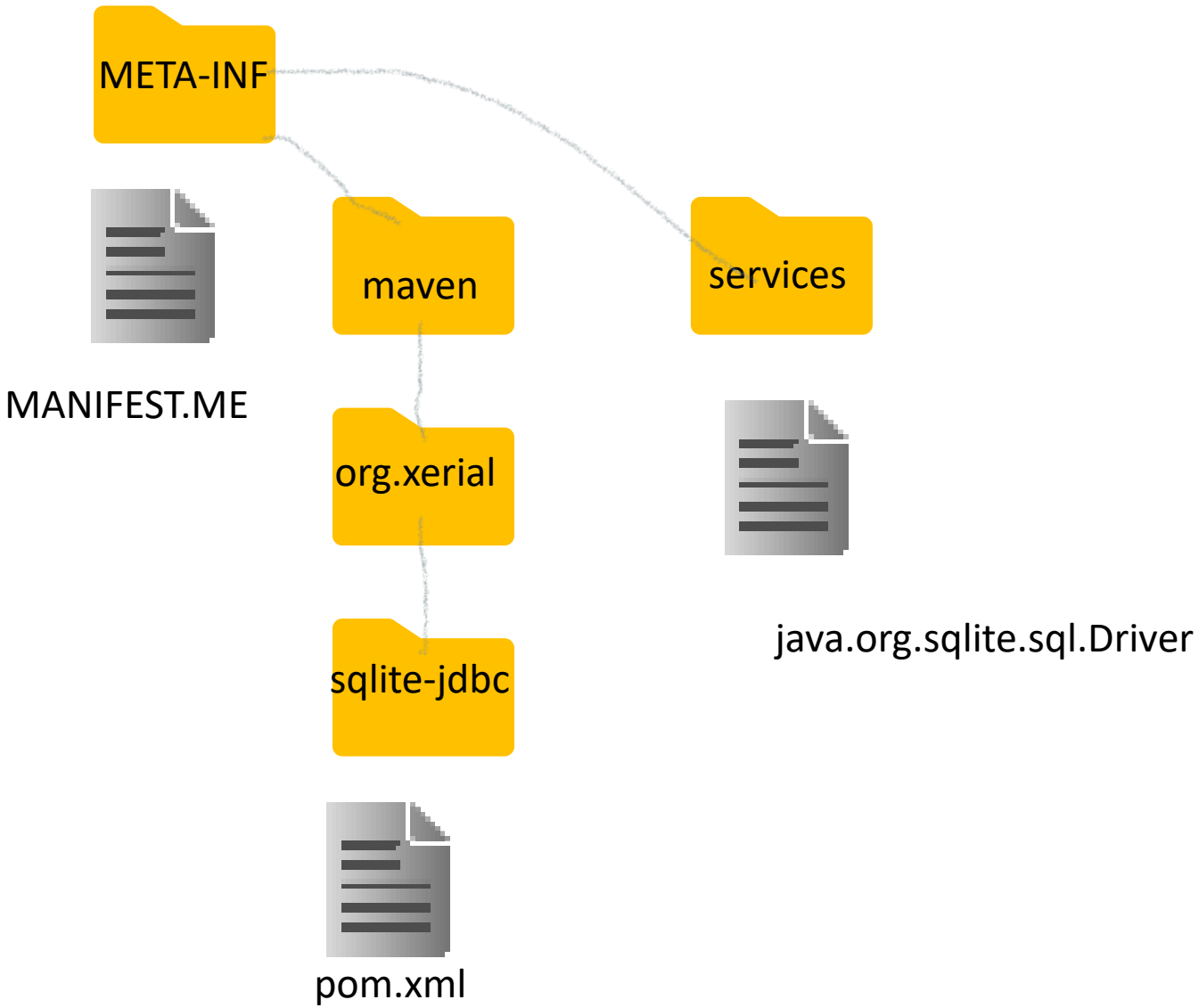
  java –jar myprogram.jar

JAR files: Java Archive

"jar" means "java archive", it's inspired by "tar" (tape archive), an old Unix command. It's also a pun, as a jar is usually a glass or earthenware container with a wide opening.

Technically, a .jar file is a compressed (zip) file.

It practice, it IS a zip file, and you can apply unzip to a .jar. Which we will do for the SQLite driver as an exercise.



This is what you find, with obvious traces of Maven. The manifest describes the contents of the .jar

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven Bundle Plugin
Built-By: leo
Build-Jdk: 1.8.0_192
Bnd-LastModified: 1577134259111
Bundle-Description: SQLite JDBC library
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0.txt
Bundle-ManifestVersion: 2
Bundle-Name: SQLite JDBC
Bundle-SymbolicName: org.xerial.sqlite-jdbc;singleton:=true
Bundle-Version: 3.30.1
Export-Package: org.sqlite;version="3.30.1.SNAPSHOT";uses:="javax.sql,
 org.sqlite.core,org.sqlite.date",org.sqlite.core;version="3.30.1.SNAP
 SHOT";uses:="org.sqlite,org.sqlite.jdbc4",org.sqlite.util;version="3.
 30.1.SNAPSHOT",org.sqlite.date;version="3.30.1.SNAPSHOT",org.sqlite.j
 avax;version="3.30.1.SNAPSHOT";uses:="javax.sql,org.sqlite,org.sqlite
 .jdbc4",org.sqlite.jdbc4;version="3.30.1.SNAPSHOT";uses:="javax.sql,o
 rg.sqlite,org.sqlite.core,org.sqlite.jdbc3",org.sqlite.jdbc3;version=
 "3.30.1.SNAPSHOT";uses:="org.sqlite,org.sqlite.core"
Import-Package: javax.sql;resolution:=optional
Originally-Created-By: Apache Maven Bundle Plugin
Tool: Bnd-2.1.0.20130426-122213
```

This is the content of the Manifest file.

Jar Files

References to files in a .jar are supposed to be in the .jar, unless they are prefixed by “file:”

<u>References to files:</u>

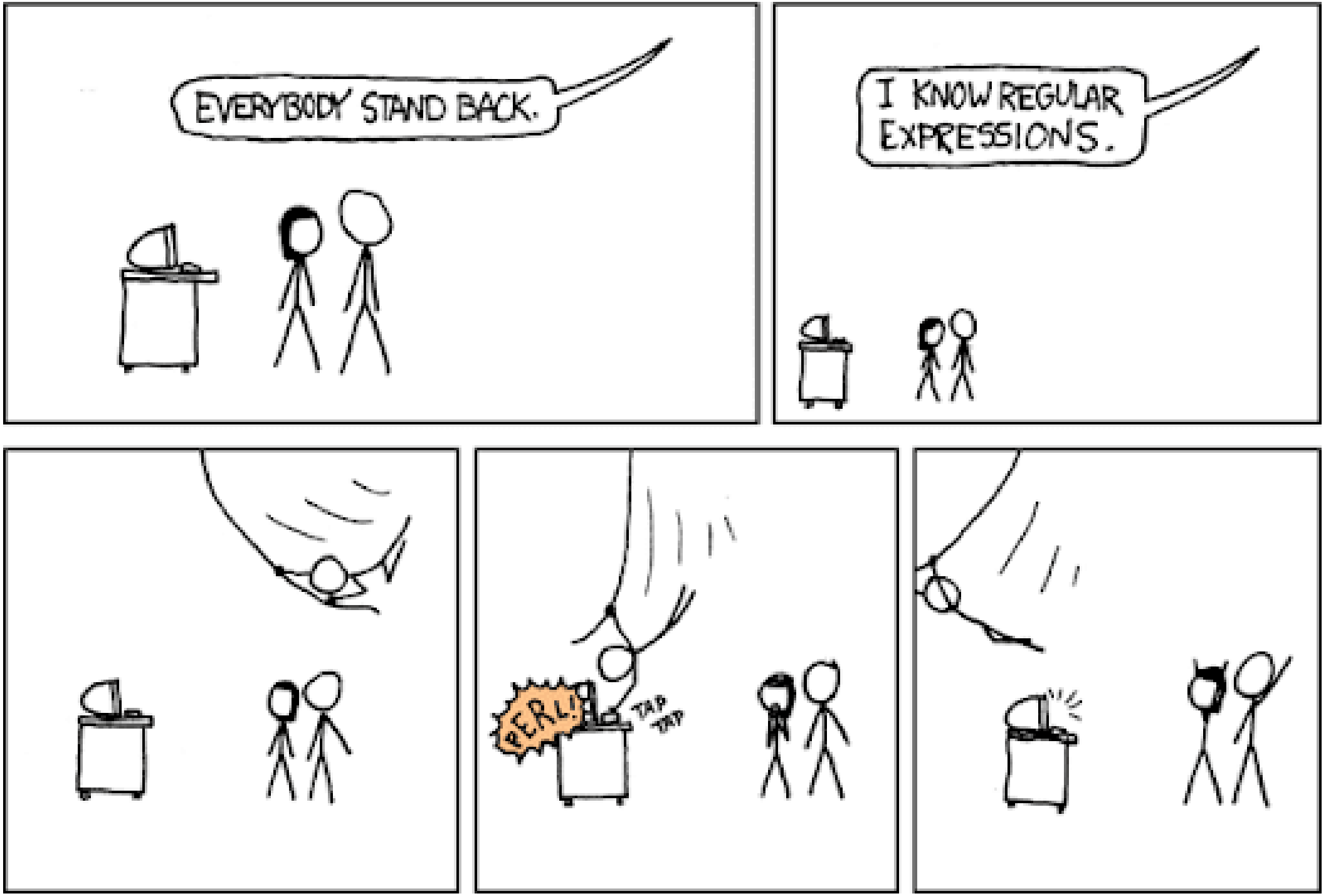
- file:path
- Operating System

look in the CLASSPATH variable

```
this.getClass()
    .getClassLoader()
    .getResource(“images/image.png”)
```

images must be in CLASSPATH

Regex


EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

New Problem Type: String Pattern Matching

- Problem description
  - **Input**: a pattern and a text
  - **Output:** Occurrences of the pattern in the text

- Applications:
  - Searching in a document
  - Finding web pages relevant to queries
  - Searching for patterns in DNA sequences

Examples of String Matching

Input:

Main String: "CODESDOPE", Pattern: "CODE"

Output:

Pattern found at location: 0

Input:

Main String: "ABAAABAAAAABBAAAABA", Pattern: "AAAB"

Output:

Pattern found at location: 2

Pattern found at location: 8

Pattern found at location: 14

New Problem Type: Regular Expressions and String Pattern Matching

In computing, a regular expression, also referred to as "regex" or "regexp", provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor.

Regular Expressions

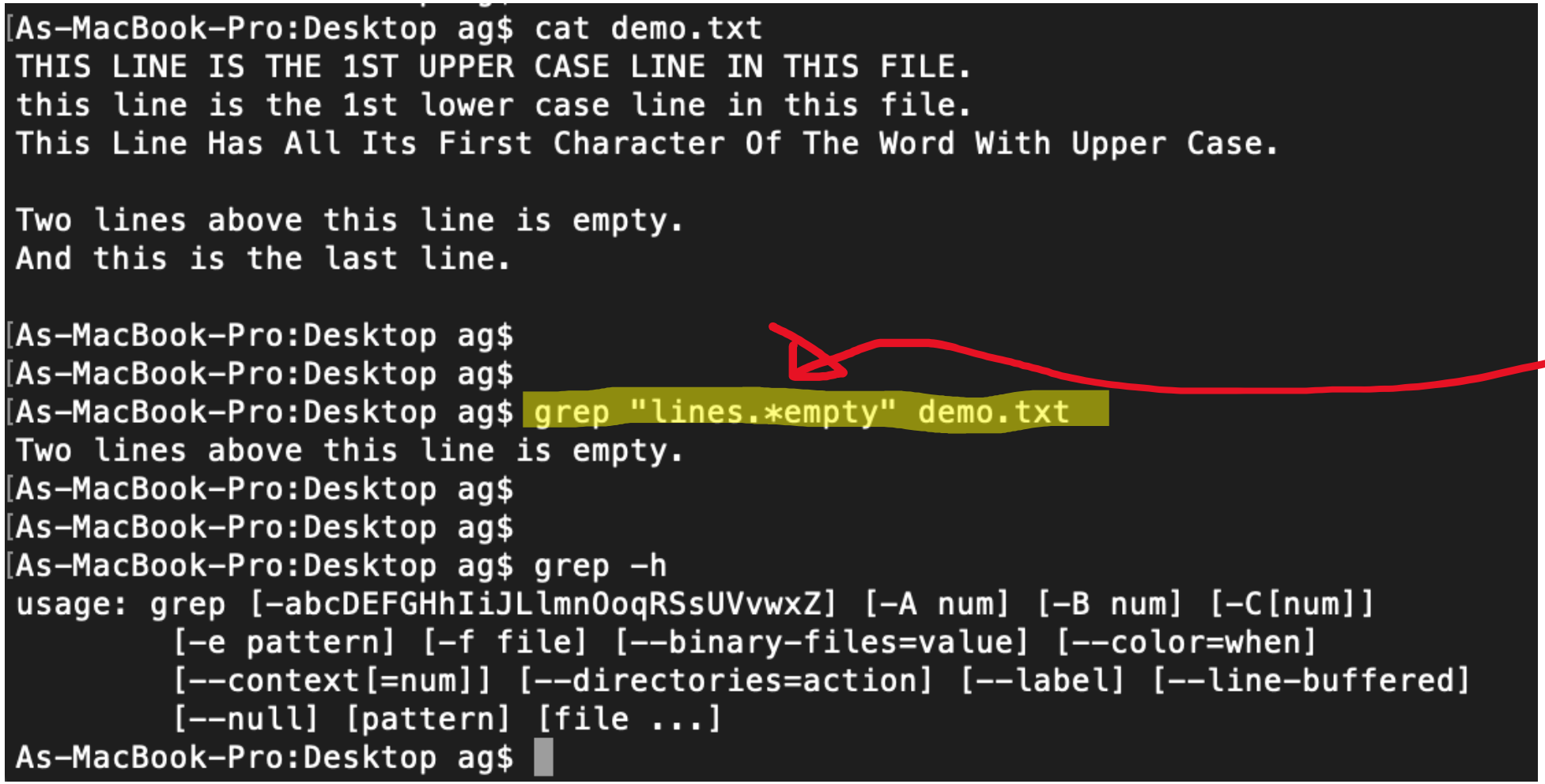
- Regular expression (also called regex or RE) is useful for describing search patterns for matching text.
- A Regex is just a sequence of some characters that defines a search pattern.
- Regex are applied widely for parsing, filtering, validating, and extracting meaningful information from large text, such as logs and output generated from other programs.

Some Background

- The term *regular expression comes from mathematics and computer science theory,* where it reflects a trait of mathematical expressions called *regularity. Such an expression* can be implemented in software using a deterministic finite automaton (DFA). A DFA is a finite state machine that doesn't use backtracking.
- The text patterns used by the earliest *grep tools were regular expressions in the mathematical* sense. Though the name has stuck, modern-day Perl-style regular expressions are not regular expressions at all in the mathematical sense.
- They're implemented with a nondeterministic finite automaton (NFA). All a practical programmer needs to remember from this note is that some ivory tower computer scientists get upset about their well-defined terminology being overloaded with technology that's far more useful in the real world.

GREP

file



Regex: grep returns lines matching the regex pattern

Formal Languages

Some definitions…

- An alphabet is a set of symbols
- A string is a finite sequence of alphabet symbols
- A formal language is a set of strings (possibly infinite if repeated) over the same alphabet

Languages

- Σ^* : ”sigma star” denotes the set of finite length strings formed from characters in the alphabet Σ
- ϵ : “epsilon” is the empty string and it also belongs to Σ^*

Example: we have $\Sigma = \{0,1\}$ is the alphabet of binary strings with 0’s and 1’s

- A language is a subset of Σ^*

$L = \{\epsilon, 0, 1, 00, 01, 10, 000, 001, 010, 100, 101, 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010, \ldots\}$

formal language	*in the language*	*not in the language*
second-to-last symbol is a	aa bbbab bbbbbbbbbababab	a aaaba bbbbbbbbbbbbb
equal numbers of a*s and* b*s*	ba bbaaba aaaabbbbbbbaaaba	a bbbaa abababababababa
palindromes	a aba abaabaabaaba	ab bbbba abababababababab
contain the pattern abba	abba abaababbabbababbba bbbbbbbbbbabbabbbbb	abb bbabaab aaaaaaaaaaaaaaaaa
number of b*s is divisible by 3*	bbb baaaaabaaaab bbbabbaaabaaababababaaa	bb abababab aaaaaaaaaaaaaaaab

Examples of formal languages over a binary alphabet

More Examples of Formal Languages

	symbols	*symbol name*	*string name*
binary	01 (or ab)	bit	bitstring
Roman	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	letter	word
decimal	0123456789	digit	integer
special	~`!@#$%^&*()_-+={[}]|\:;"'<,>.?/		
keyboard	*Roman + decimal + special*	keystroke	typescript
genetic code	ATCG	nucleotide base	DNA
protein code	ACDEFGHIKLMNPQRSTVWY	amino acid	protein
ASCII	see SECTION 6.1	byte	`String`
Unicode	see SECTION 6.1	char	`String`

Commonly used alphabets and associated terminology

formal language	*in the language*	*not in the language*
amino acid encodings	AAA AAC AAG AAT ACA ACC ACG ACT TAC TAT TGC TGG TGT	TAA TAG TGA AAAAAAAAA ABCDE
U.S. telephone number	(609) 258-3000 (800) 555-1212	(99) 12-12-12 2147483648
English words	and middle computability	abc niether misunderestimate
legal English sentences	This is a sentence. I think I can.	xya a b.c.e?? Cogito ergo sum.
legal Java identifiers	a class $xyz3_XYZ	12 123a a((BC))*
legal Java programs	public class Hi { public static void main(String[] args) { } }	int main(void) { return 0; }

Regular expressions. A *regular expression (RE)* is a string of symbols that specifies a formal language. We define what regular expressions are (and what they mean) recursively, using the union, concatenation, and closure operations on sets, along with parentheses for operator precedence. Specifically, every regular expression is either an *alphabet symbol*, specifying the singleton set containing that symbol, or composed from the following operations (where R and S are REs):

- *Union:* R | S, specifying the union of the sets R and S,
- *Concatenation:* RS, specifying the concatenation of the sets R and S,
- *Closure:* R*, specifying the closure of the set R,
- *Parentheses:* (R), specifying the same set as R.

Definition. A language is *regular* if and only if it can be specified by an RE.

Using Regular Expressions to Make Many Text Processing Problems Solvable Quickly and Easily

- You might wonder why we would ever need to use regular expressions so why learn about them?
- Here are some use cases:
  - Searching for text where we don’t know the specific text we are looking for up front but we do know some rules or patterns in the text:
    - Search for an IP or MAC address in a log (e.g. web server access log)
    - Search for a 10 digit mobile number that may optionally be preceeded with a country code
  - Searching where the length of the text to extract is not known beforehand:
    - Search for URLs that start with **http://** or **https://**

Using Regular Expressions to Make Many Text Processing Problems Solvable Quickly and Easily

- Generating tokens by splitting a given text on delimiters of a variable type and length
- Extracting text that lies between 2 or more search patterns
- Validating input from the user (account number, usernames, credit card number etc)
- Getting parts of a text with some properties such as repeated words
- Conversions to custom predefined formats: e.g. insert comma after every three digits in numbers or remove commas that occur in parantheses
- Doing a global search replace while skipping escaped characters

Pattern matching

Pattern matching problem. Is a given string an element of a given set of strings?

Example 1 (from computational biochemistry)

An amino acid is represented by one of the characters CAVLIMCRKHDENQSTYFWP.

A protein is a string of amino acids.

A C_2H_2-type zinc finger domain signature is

- C followed by 2, 3, or 4 amino acids, followed by
- C followed by 3 amino acids, followed by
- L, I, V, M, F, Y, W, C, or X followed by 8 amino acids, followed by
- H followed by 3, 4, or 5 amino acids, followed by H.

Q. Is this protein in the C_2H_2-type zinc finger domain?

A. Yes.

Pattern matching

Example 2 (from commercial computing)

An e-mail address is

- A sequence of letters, followed by
- the character "@", followed by
- followed by a nonempty sequence of lowercase letters, followed by the character "."
- [any number of occurrences of the previous pattern]
- "edu" or "com" (others omitted for brevity).

Q. Which of the following are e-mail addresses?

	A.
rs@cs.princeton.edu	✓
not an e-mail address	✗
wayne@cs.princeton.edu	✓
eve@airport	✗
rs123@princeton.edu	✗

Oops, need to fix description →

Challenge. Develop a precise description of the set of strings that are legal e-mail addresses.

you could search online

Pattern matching

Example 3 (from genomics)

A nucleic acid is represented by one of the letters a, c, t, or g.

A genome is a string of nucleic acids.

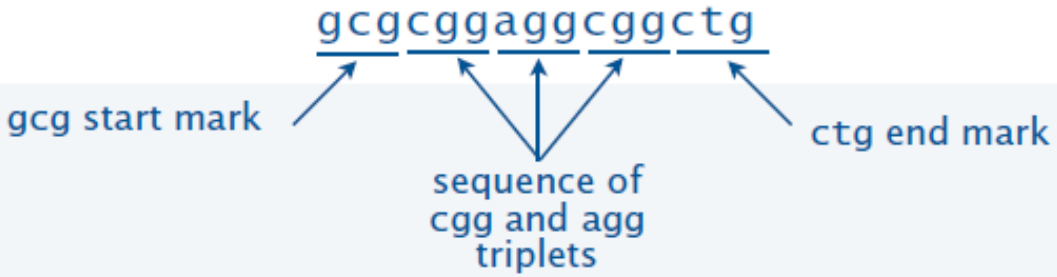
A Fragile X Syndrome pattern is a genome having an occurrence of gcg, followed by any number of cgg or agg triplets, followed by ctg.

Note. The number of triplets correlates with Fragile X Syndrome, a common cause of mental retardation.

Q. Does this genome contain a such a pattern?

gcggcgtgtgtgcgagagagtgggtttaaagctg**gcg cgg agg cgg ctg**gcgcggaggctg

A. Yes.



Regular Expressions

Clever "wild card" expressions for matching and parsing strings.

Regular Expressions are an extended pattern matching language, they correspond to Finite State Automata

Regex Example: matching an IP address

- The regular expression \b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b will match the following:
  - 110.65.147.246
  - 999.999.999.999
  - ١٢٣.१२३.০৭৯.๑๒๓
- Note: IP addresses should have numbers ranging from 0-255

Regex Example: matching an IP address

- This one will store each of the numbers in the ip address in a separate group (using ()), it will only match the numbers if they are in the range 0 – 255.

```
\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
  (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

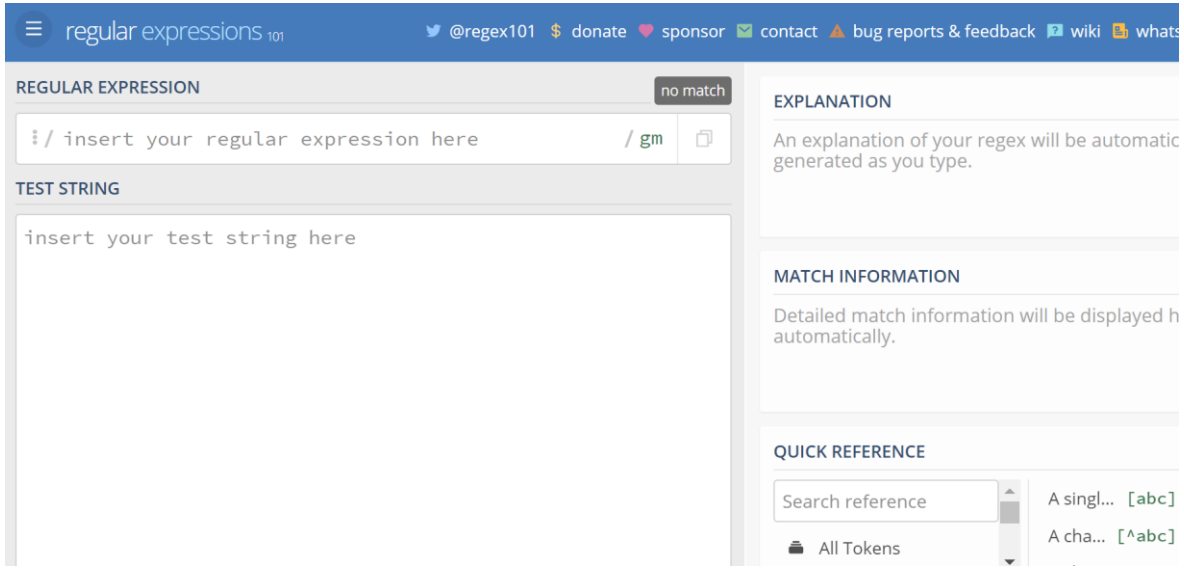
- Note: “\b” matches a word boundary, “[0-5]” matches characters 1,2,3,4 or 5, “?” means non greedy – so match one or more of the proceeding pattern

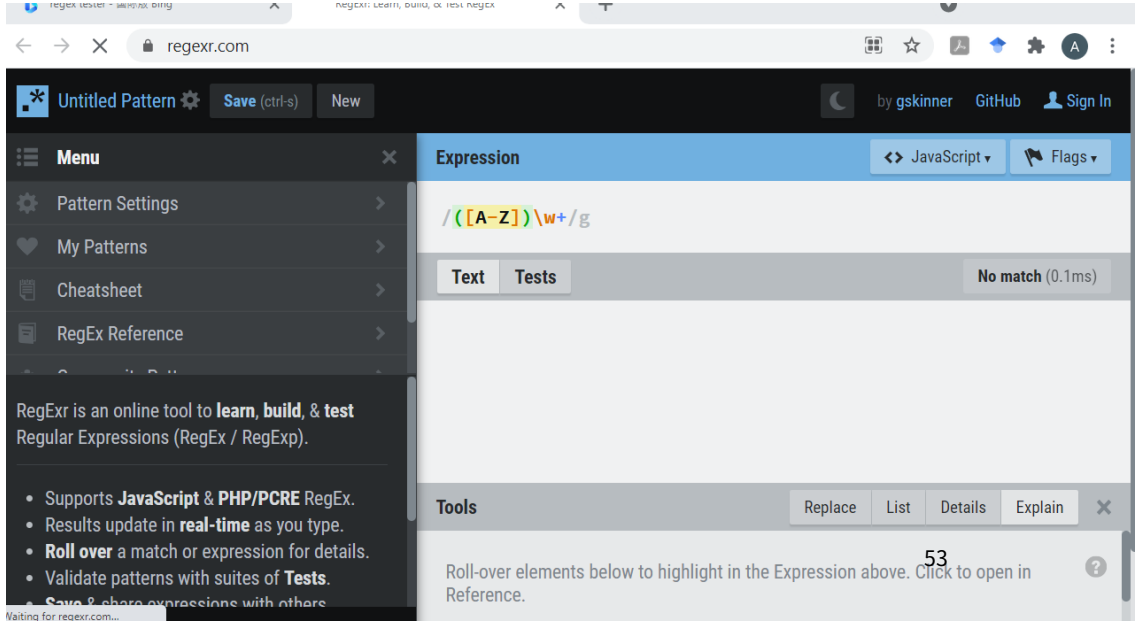
Regex Tester (also for learning regex)

https://regex101.com/

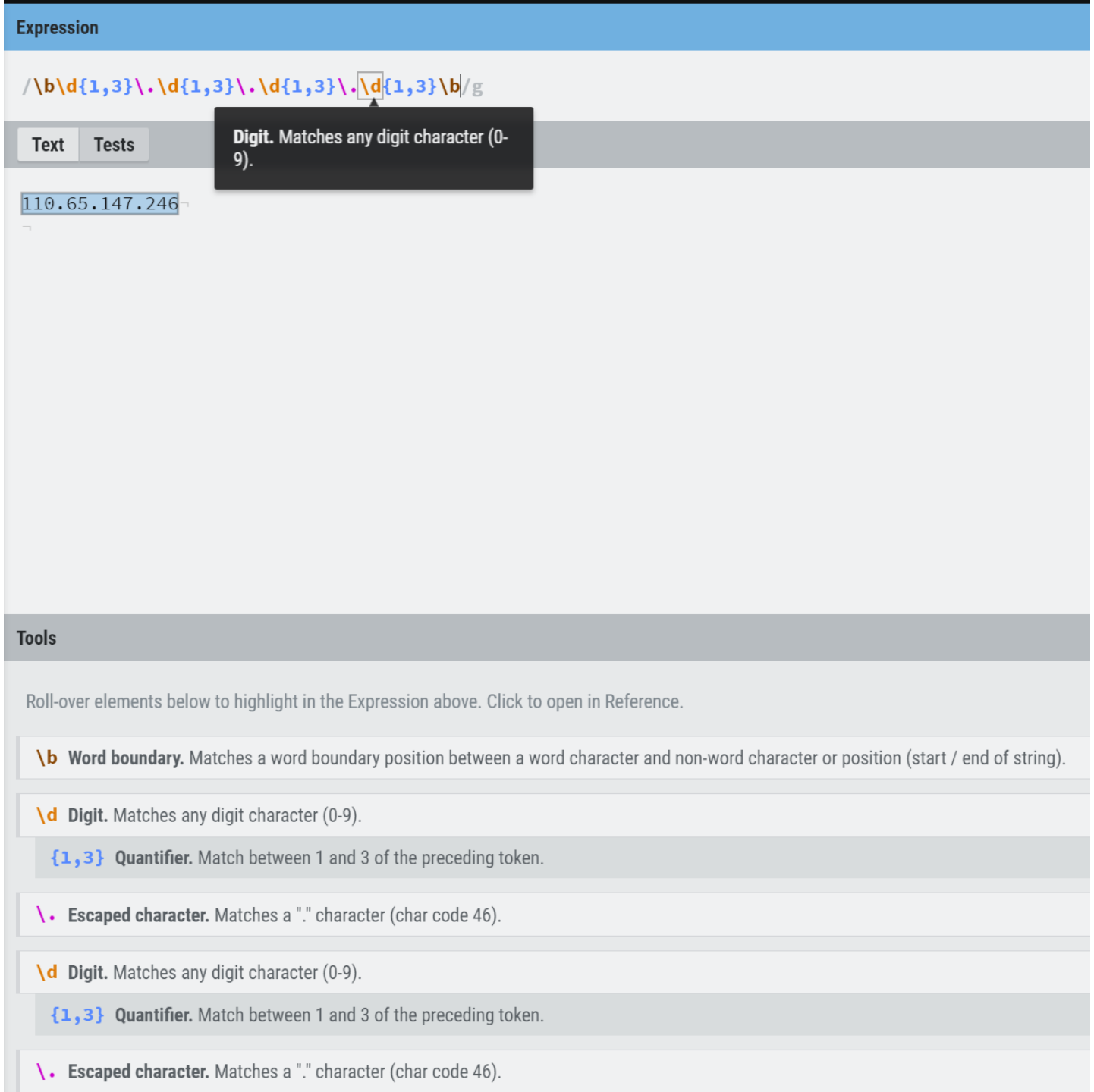
https://regexr.com/

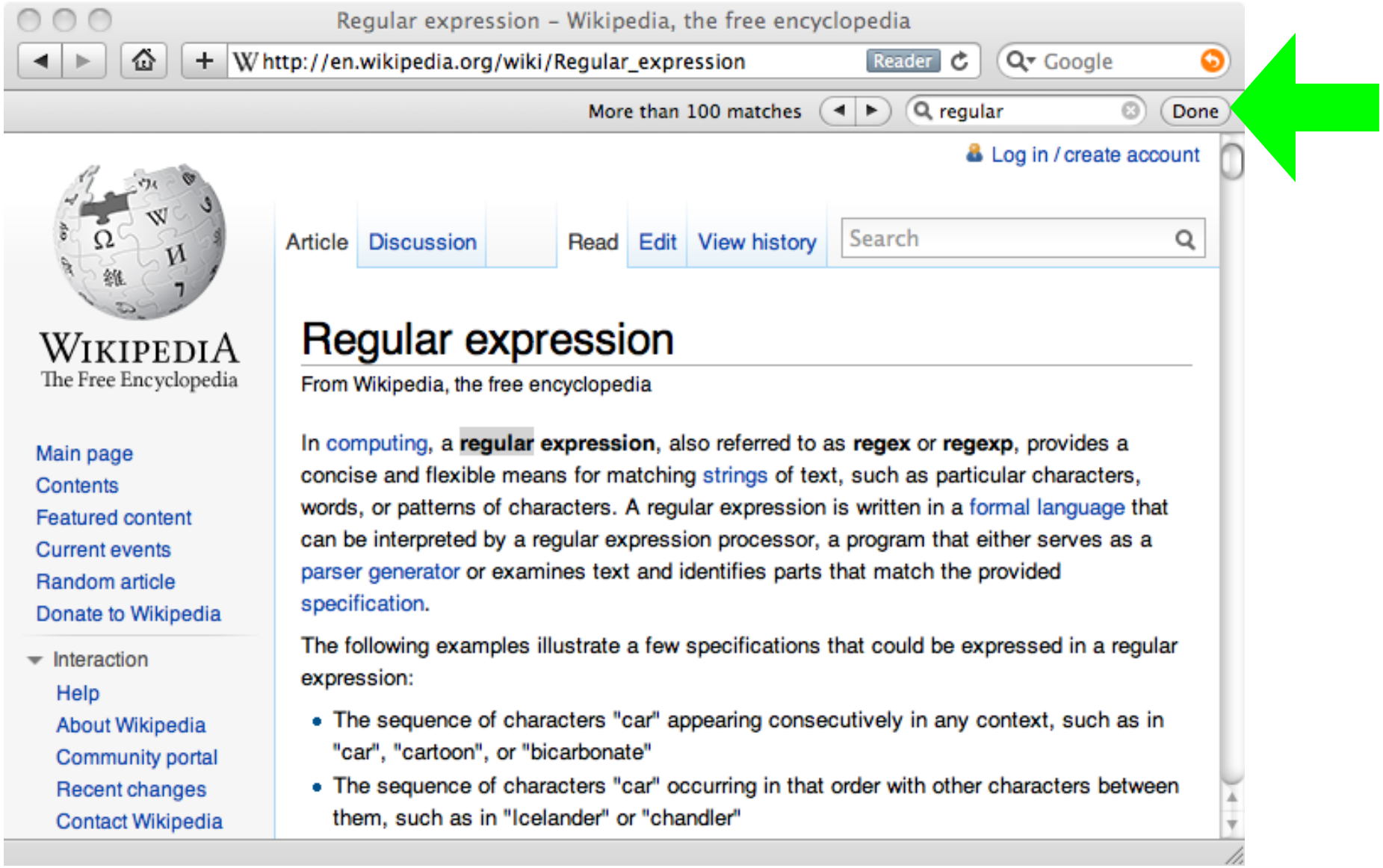
You can test regex matching easily at these links (there are others too)





https://regexr.com/





Really smart "Find" or "Search"

Understanding Regular Expressions

- Very powerful for text processing but cryptic
- Fun once you understand them
- Regular expressions are a language by themselves
- Programming with characters

http://xkcd.com/208/

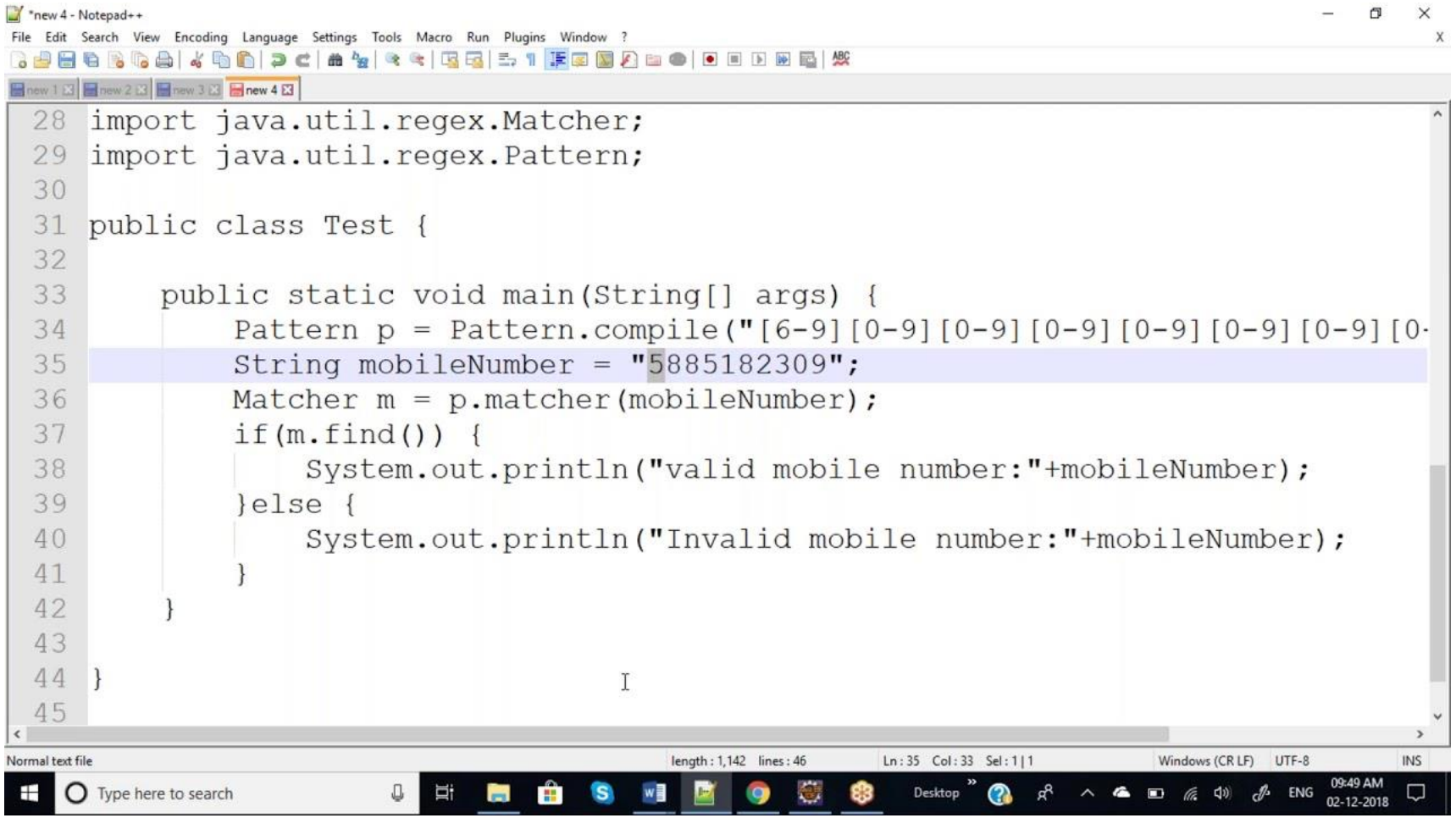
Regular Expression Quick Guide

^ Matches the beginning of a line
$ Matches the end of the line
. Matches any character
\s Matches whitespace
\S Matches any non-whitespace character
* Repeats a character zero or more times
*? Repeats a character zero or more times (non-greedy)
+ Repeats a chracter one or more times
+? Repeats a character one or more times (non-greedy)
[aeiou] Matches a single character in the listed set
[^XYZ] Matches a single character not in the listed set
[a-z0-9] The set of characters can include a range
(Indicates where string extraction is to start
) Indicates where string extraction is to end

Using Regular Expressions in Your Programs

Before you can use regular expressions in your program, you must import the library java.util.regex.Matcher and Pattern libraries, they are used in a specific way in java, and regular expressions are used in many other programming languages (python, etc)

Java


*new 4 - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
new 1 new 2 new 3 new 4
28 import java.util.regex.Matcher;
29 import java.util.regex.Pattern;
30
31 public class Test {
32
33 public static void main(String[] args) {
34 Pattern p = Pattern.compile("[6-9][0-9][0-9][0-9][0-9][0-9][0-9][0-
35 String mobileNumber = "5885182309";
36 Matcher m = p.matcher(mobileNumber);
37 if(m.find()) {
38 System.out.println("valid mobile number:"+mobileNumber);
39 }else {
40 System.out.println("Invalid mobile number:"+mobileNumber);
41 }
42 }
43
44 }
45
Normal text file
length : 1,142 lines : 46
Ln : 35 Col : 33 Sel : 1 | 1
Windows (CR LF)
UTF-8
INS
Type here to search
Desktop
ENG
09:49 AM
02-12-2018

Wild-Card Characters

- The dot character matches any character
- If you add the asterisk character, the character is "any number of times"

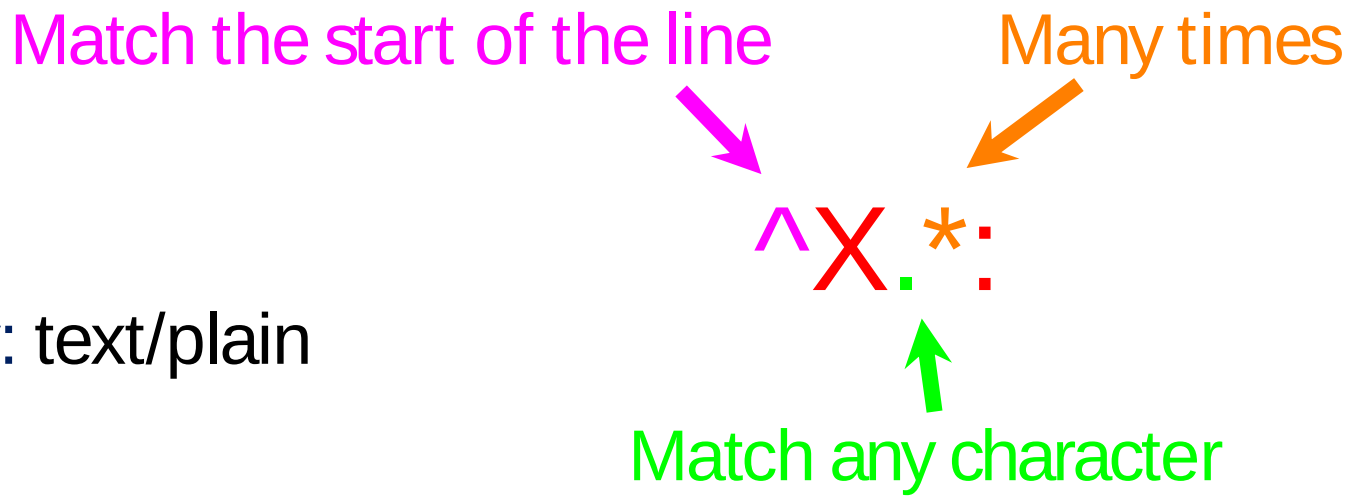
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body: text/plain

^X.*:

Wild-Card Characters

- The dot character matches any character
- If you add the asterisk character, the character is "any number of times"

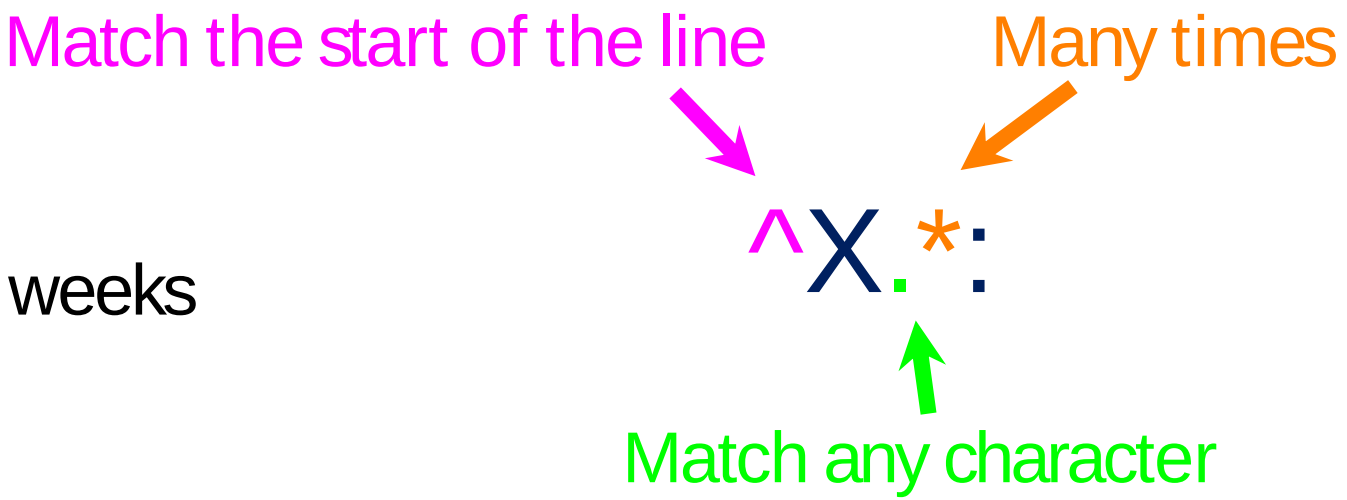
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body: text/plain



Fine-Tuning Your Match

- Depending on how "clean" your data is and the purpose of your application, you may want to narrow your match down a bit

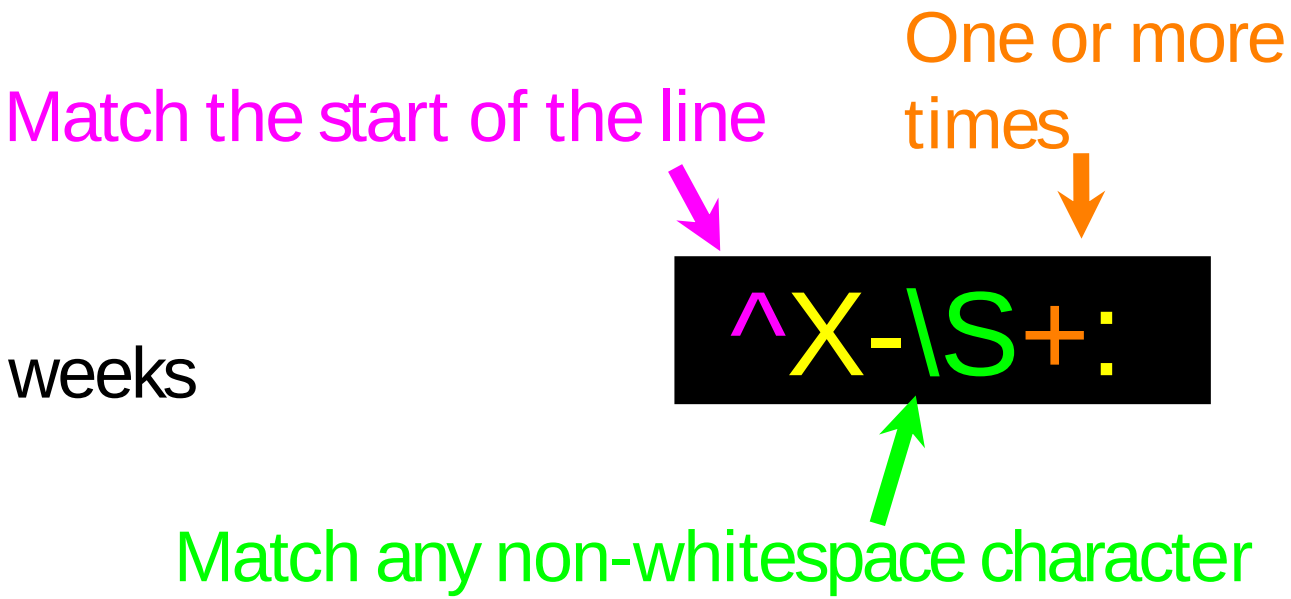
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks



Fine-Tuning Your Match

- Depending on how "clean" your data is and the purpose of your application, you may want to narrow your match down a bit

X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks



Matching and Extracting Data

We can extract the strings using () and group

```
String x = "My 2 favorite numbers are 19 and 42";
String patS  = "([0-9]+)";
Pattern p = Pattern.compile(patS );
Matcher m = p.matcher(x);
if (m.find())
        System.out.println(m.group(0)+" "+ m.group(1)+" "
                m.group(2));
```

[0-9]+

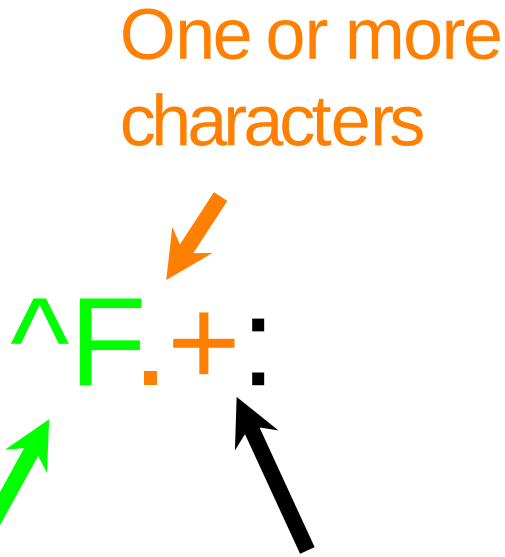
One or more digits

2 19 42

Warning about Greedy Matching

- The repeat characters (* and +) push outward in both directions (greedy) to match the largest possible string

String patS = ”^F.+:”;
String = ”From: Using the : character”
Will match this
['From: Using the :']



Why not 'From:'?

First character in the match is an F

Last character in the match is a :

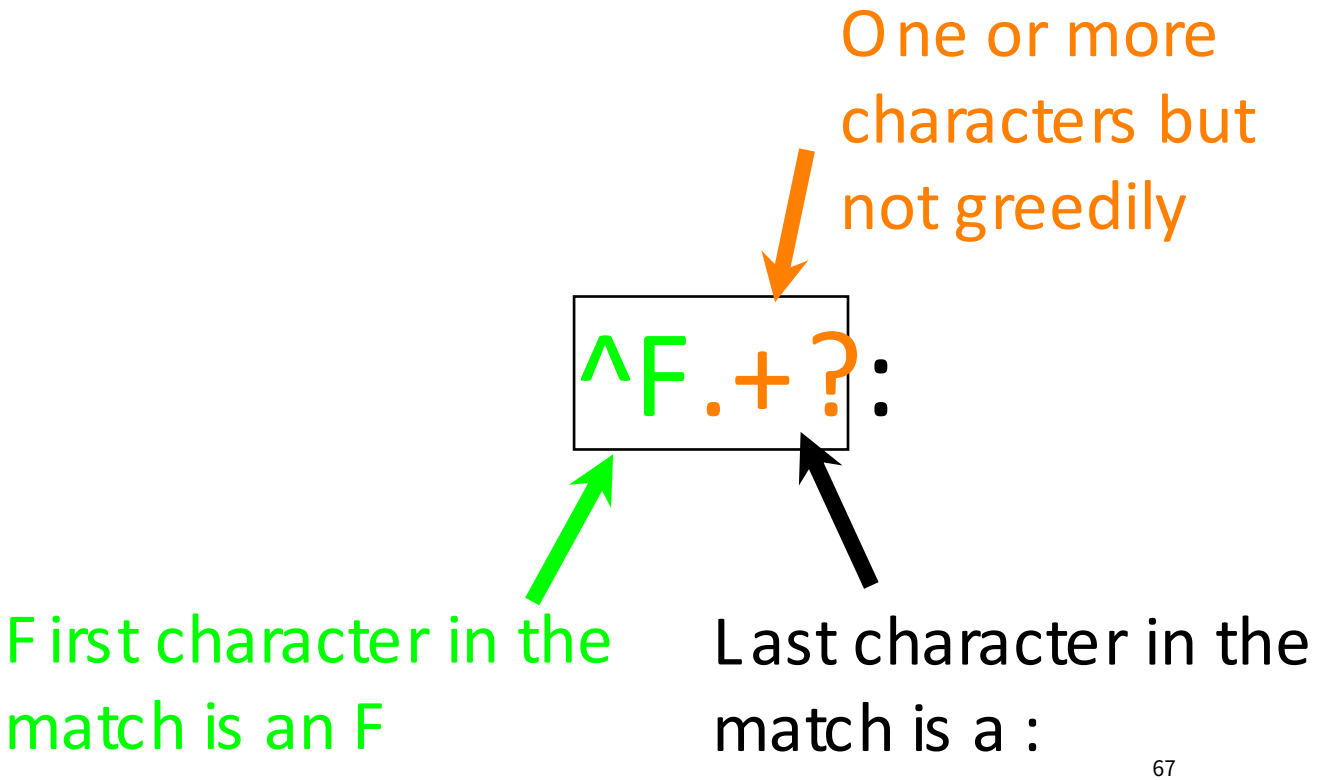
Non-Greedy Matching

- Not all regular expression repeat codes are greedy! If you add a ? character - the + and * chill out a bit...

Line: 'From: Using the : character'

Pattern '^F.+?:'

Matches: ['From:']



Regular Expression Quick Guide

^ Matches the beginning of a line
$ Matches the end of the line
. Matches any character
\s Matches whitespace
\S Matches any non-whitespace character
* Repeats a character zero or more times
*? Repeats a character zero or more times (non-greedy)
+ Repeats a chracter one or more times
+? Repeats a character one or more times (non-greedy)
[aeiou] Matches a single character in the listed set
[^XYZ] Matches a single character not in the listed set
[a-z0-9] The set of characters can include a range
(Indicates where string extraction is to start
) Indicates where string extraction is to end

(Deterministic) Finite Automata

- May be used to define languages formally using:
  - A set of states
  - An alphabet
  - A transition function
  - A starting state
  - A final state (that is also called an accepting state)

Transition function

- Transition functions take 2 arguments: a state and an input symbol:

Formally:

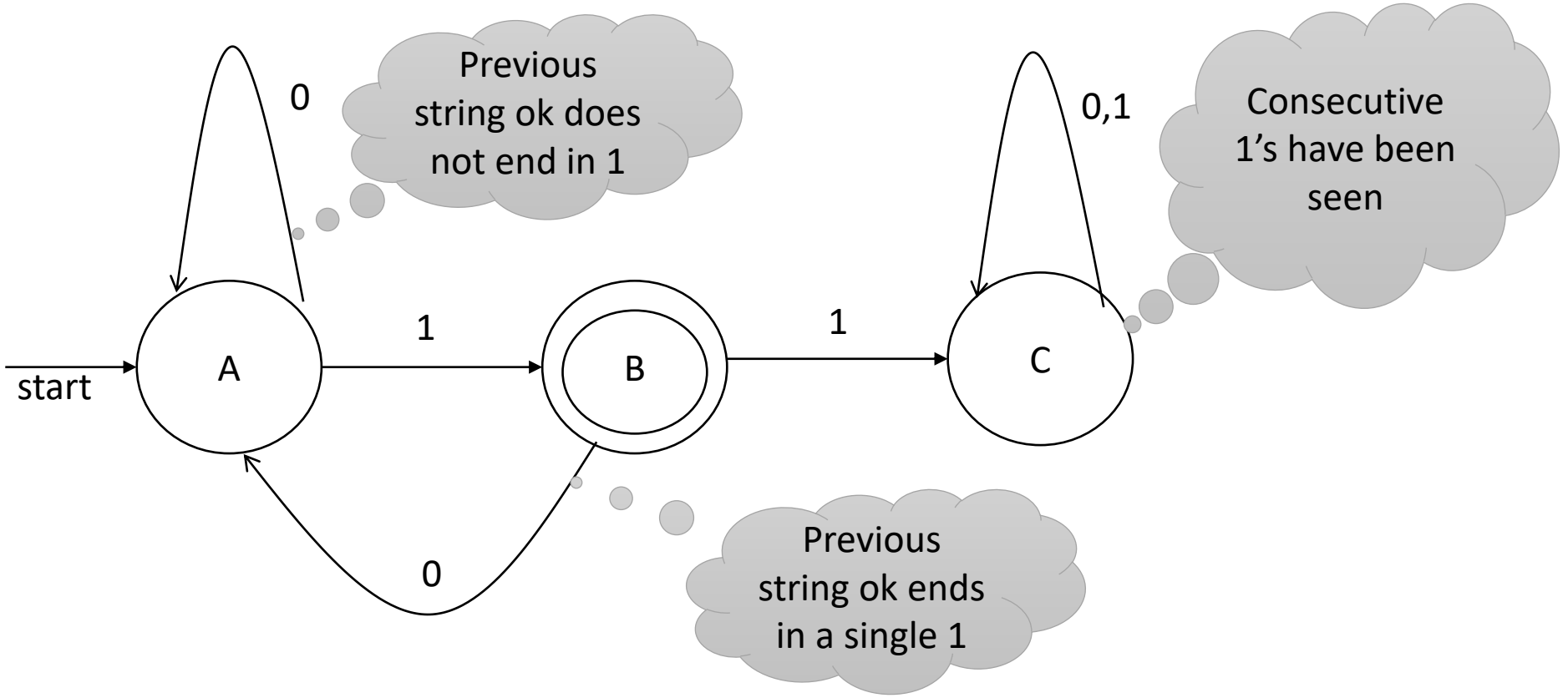
$$\delta(q, a) = q'$$

- δ will map the state q to another state q’ given the input symbol a

FSM Graphs

- Vertices represent states
- Edges are directed and represent a transition function
  - An edge from state q to q’ is labelled by input symbols that have transitions from q to q’
- An arrow should be labelled start to indicate the starting state
- The final state can be marked by a double circle

Example for alphabet {0,1}

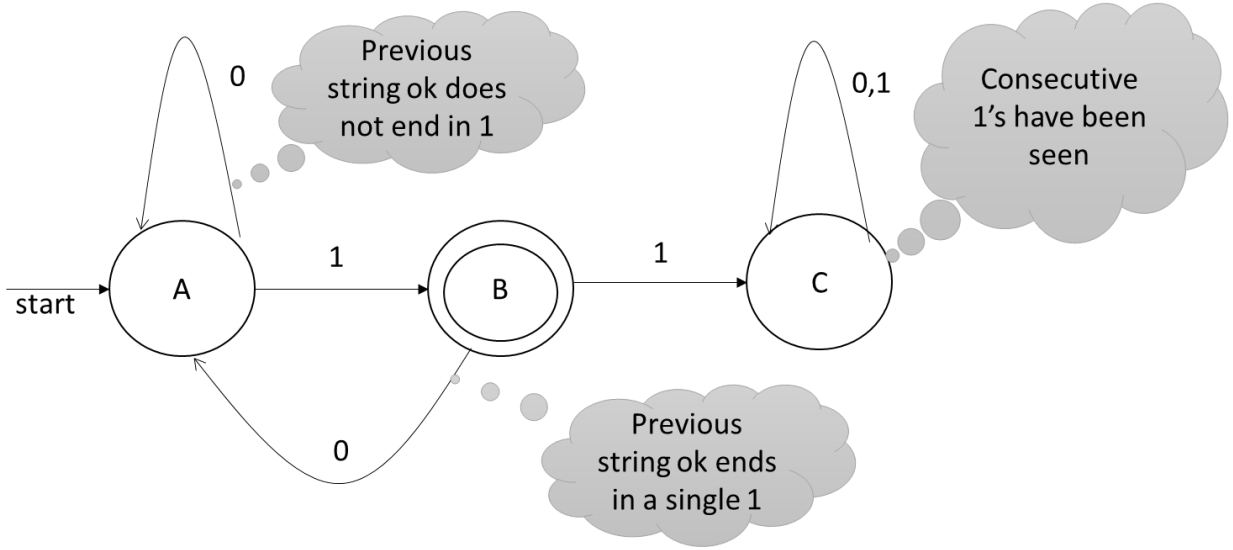


This FA accepts all strings without two consecutive 1's

Transition Tables

Next state from a for input 0 is A

State	0	1
A	A	B
B	A	C
C	C	C



$$\delta(A,0) = A$$
$$\delta(A,1) = B$$
$$\delta(B,0) = A$$
$$\delta(B,1) = C$$
$$\delta(C,0) = C$$
$$\delta(C,1) = C$$

2 Types of Finite Automata

- Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time

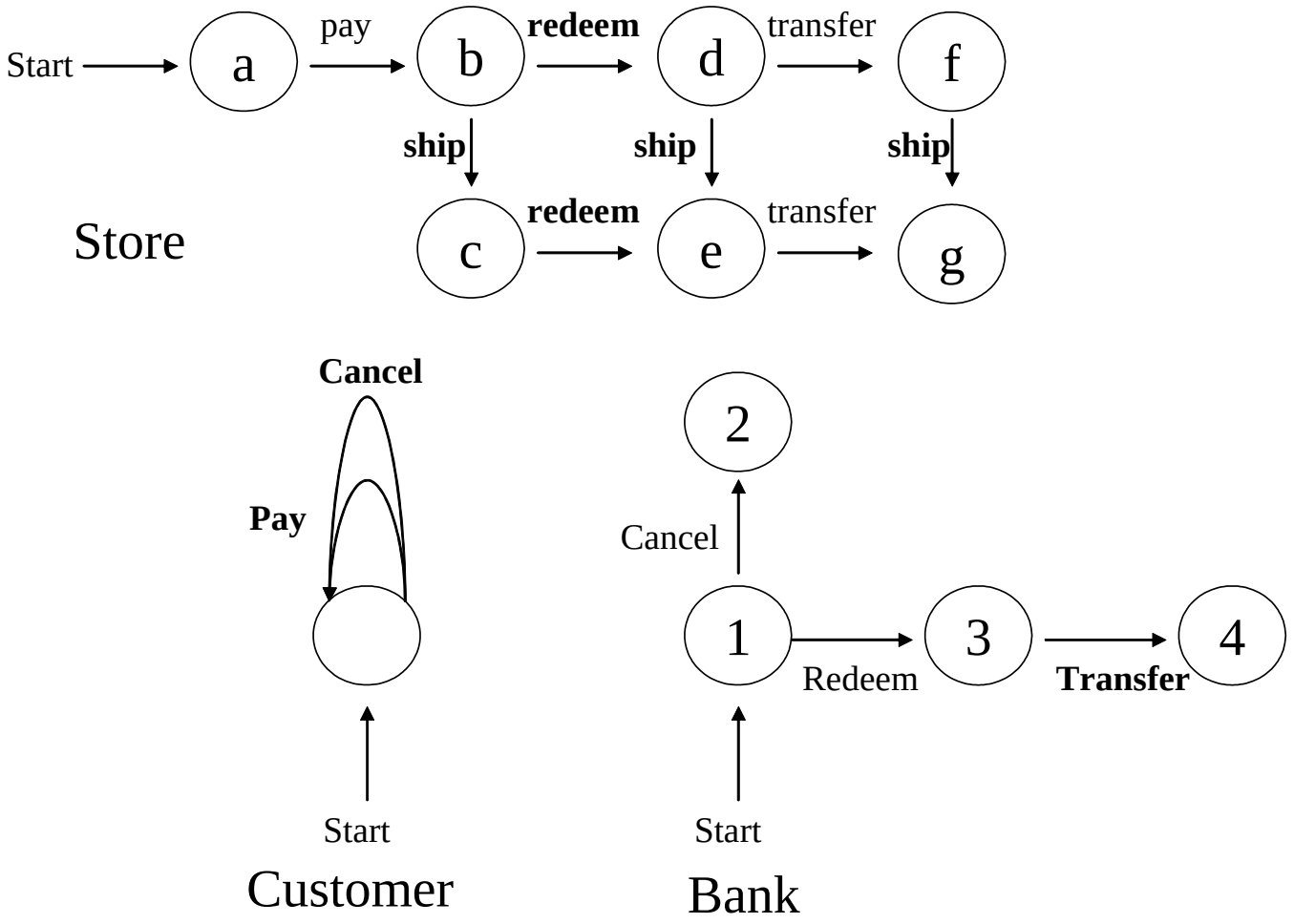
- Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time

Many Problems Can Be Modelled With FA

- Customer shopping at a store with an electronic transaction with the bank
  - The customer may *pay* the e-money or *cancel* the e-money at any time.
  - The store may *ship* goods and *redeem* the electronic money with the bank.
  - The bank may *transfer* any redeemed money to a different party, say the store.

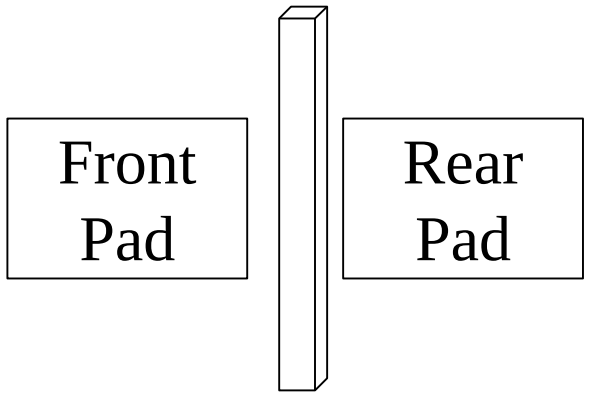
- Can model this problem with three automata

Bank Automata



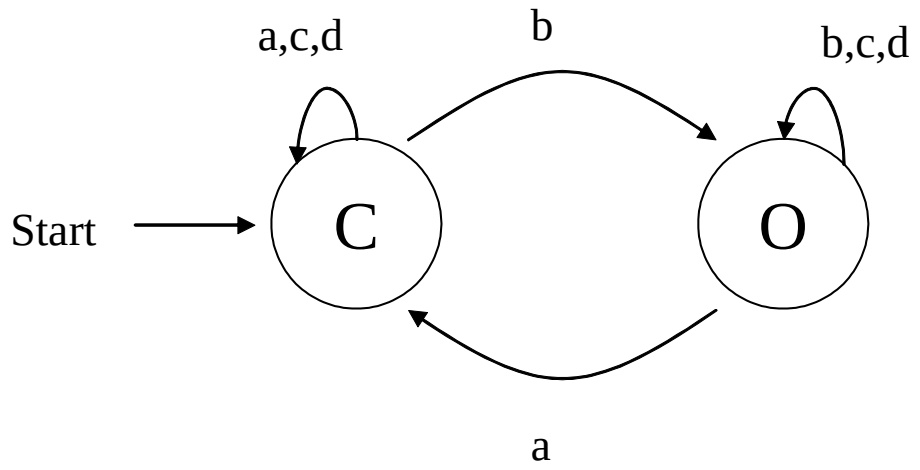
Simple Example – 1 way door

- As an example, consider a one-way automatic door. This door has two pads that can sense when someone is standing on them, a front and rear pad. We want people to walk through the front and toward the rear, but not allow someone to walk the other direction:



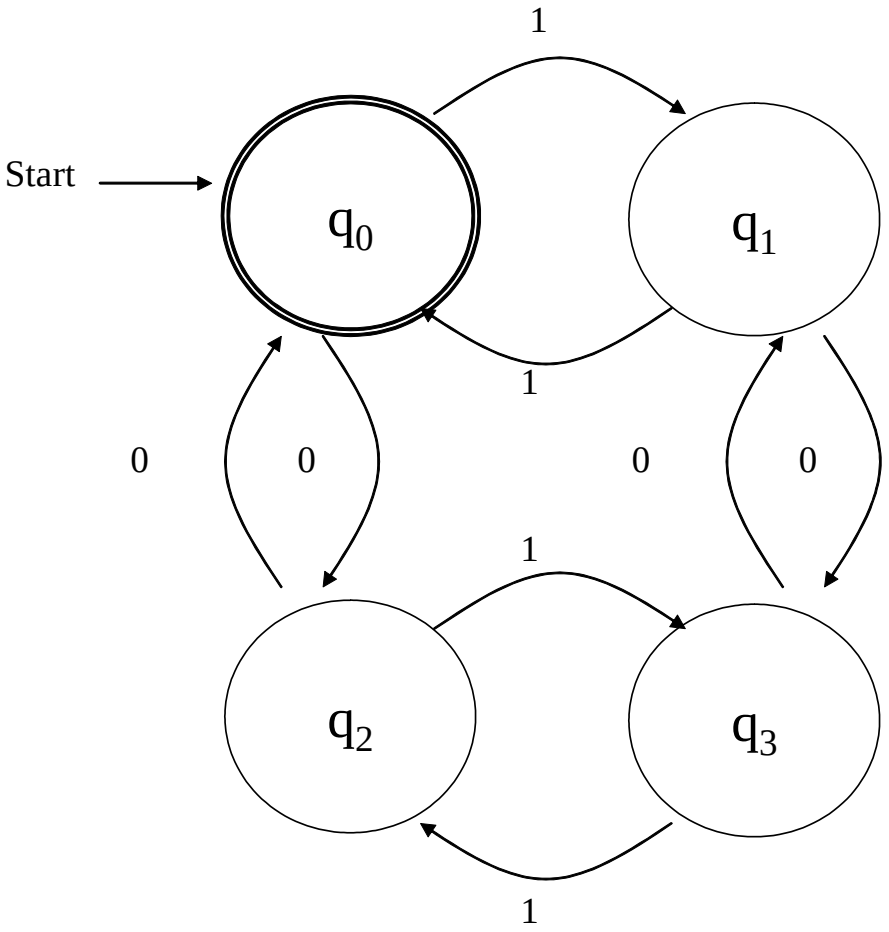
One Way Door

- Let's assign the following codes to our different input cases:
  - a - Nobody on either pad
  - b - Person on front pad
  - c - Person on rear pad
  - d - Person on front and rear pad
- We can design the following automaton so that the door doesn't open if someone is still on the rear pad and hit them:



DFA Example

Here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even



NFA

- Non Deterministic Finite Automata allow several states to be present at once
- Any NFA can be shown to have an equivalent to DFA
- We will not further discuss NFA in this course

Matching with Finite Automata

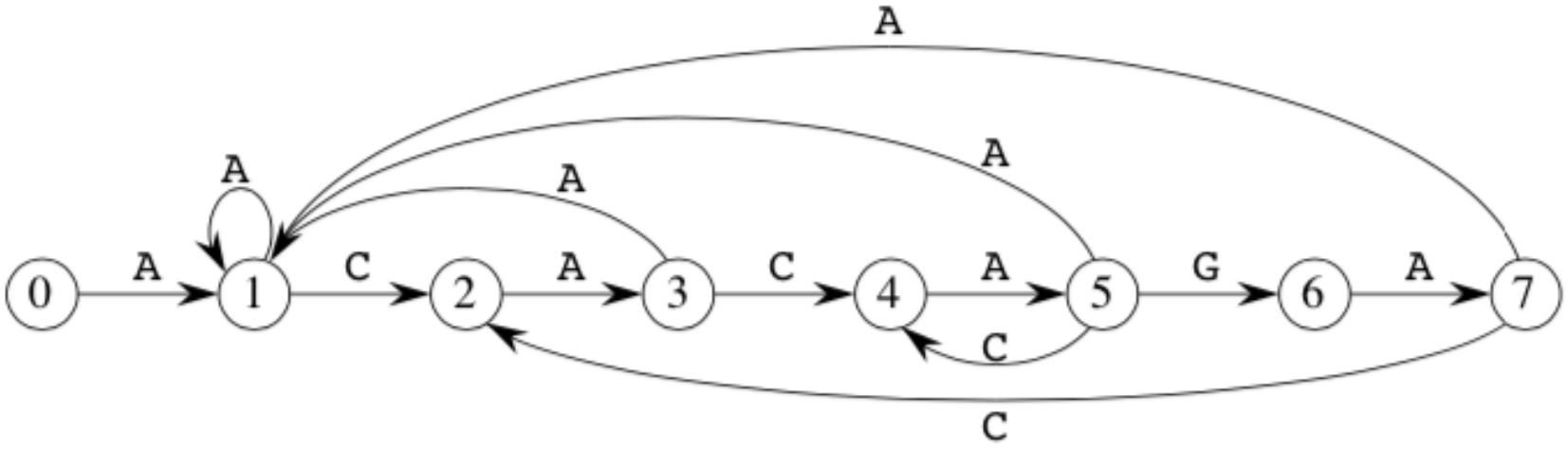
- A ***finite automaton (FA)*** is a set of states and a way to go from state to state based on a sequence of input characters. An FA starts in a given state and "consumes" characters one at a time. Based on the state it's in and the character just consumed, it moves to a new state.

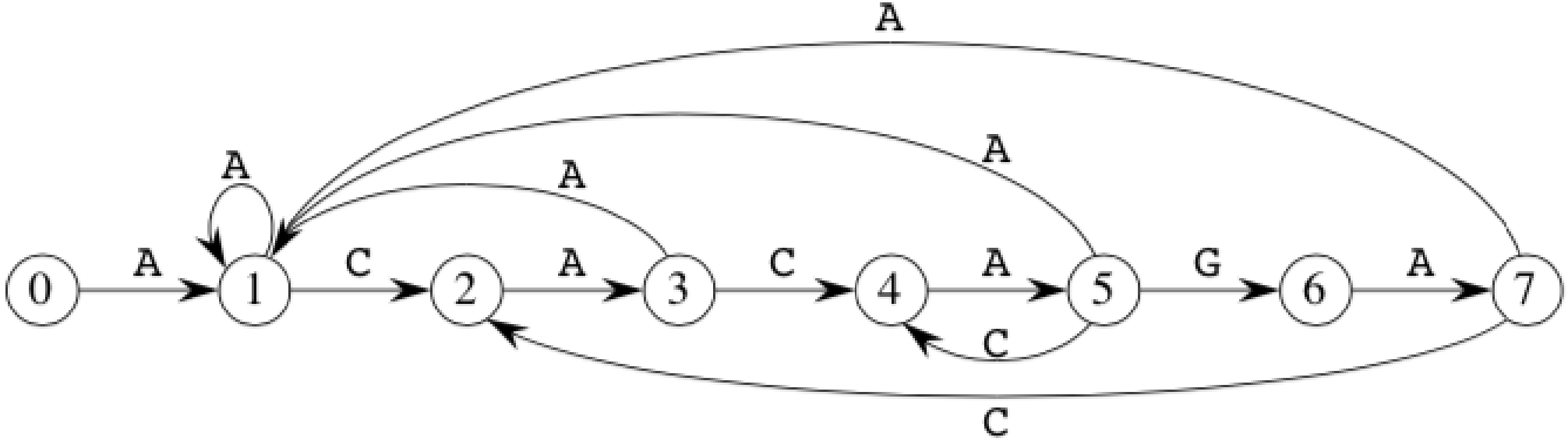
Formally: A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function $Q \times \Sigma \to Q$, called the ***transition function*** of M.

FA Example

- Consider DNA sequencing where an alphabet Σ = {A, C, G, T}
- Pattern P = ACACAGA
- The Finite Automata is:





- The horizontal main line has edges labelled with P
- Whenever P occurs in the text the FA moves right to the next state
- When it reaches the far right hand state we have a match

- Note: the transition function δ (delta) is defined for all states $q \in Q$ and all characters $a \in \Sigma$ (in the alphabet) and missing arrows are assumed to be transitions to state 0

FA-MATCHER(T, δ, n, m)

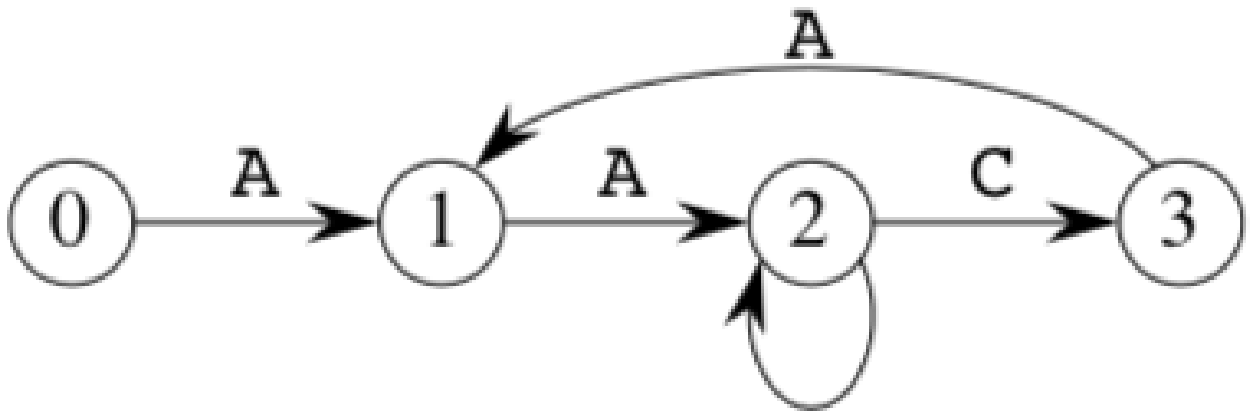
```
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Pattern occurs with shift" i − m
```

Example 2

Sequence of states, input characters, and output shifts:

i		1	2	3	4	5	6	7	8	9	10	11	12	13	
character		G	T	A	A	C	A	G	T	A	A	A	C	G	
state	0	0	0	0	1	2	3	1	0	0	1	2	2	3	0
output shift							2							9	

Example: $P = \texttt{AAC}, T = \texttt{GTAACAGTAAACG}$. FA is



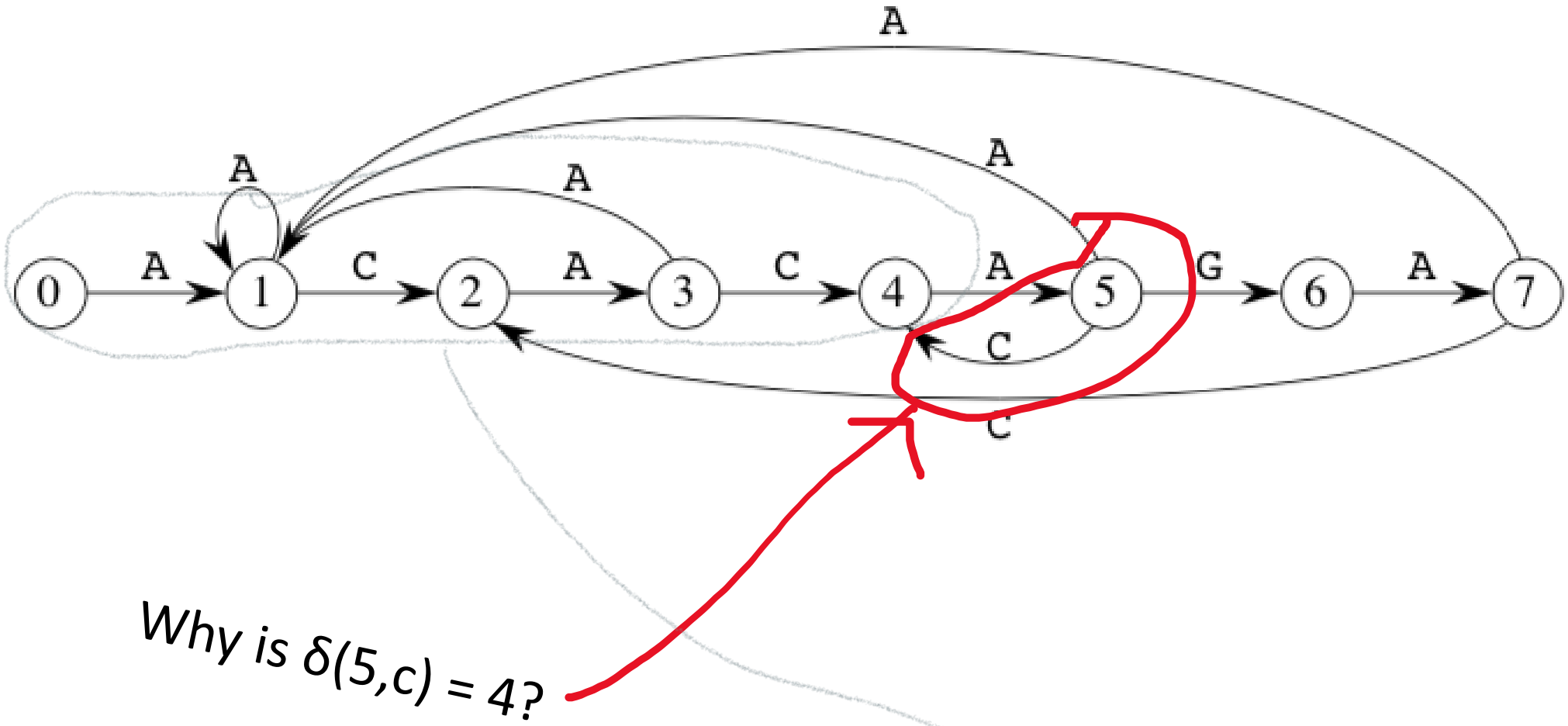
Θ(n)

The pre-process to build the Finite Automaton

- We know it has states 0 .. M (the length of the pattern)
- We know the starting state is 0
- We “just” need to determine the transition function δ

- **Basic idea**: when the FA is in state k, the most recent characters read from the text are the first k characters in the pattern

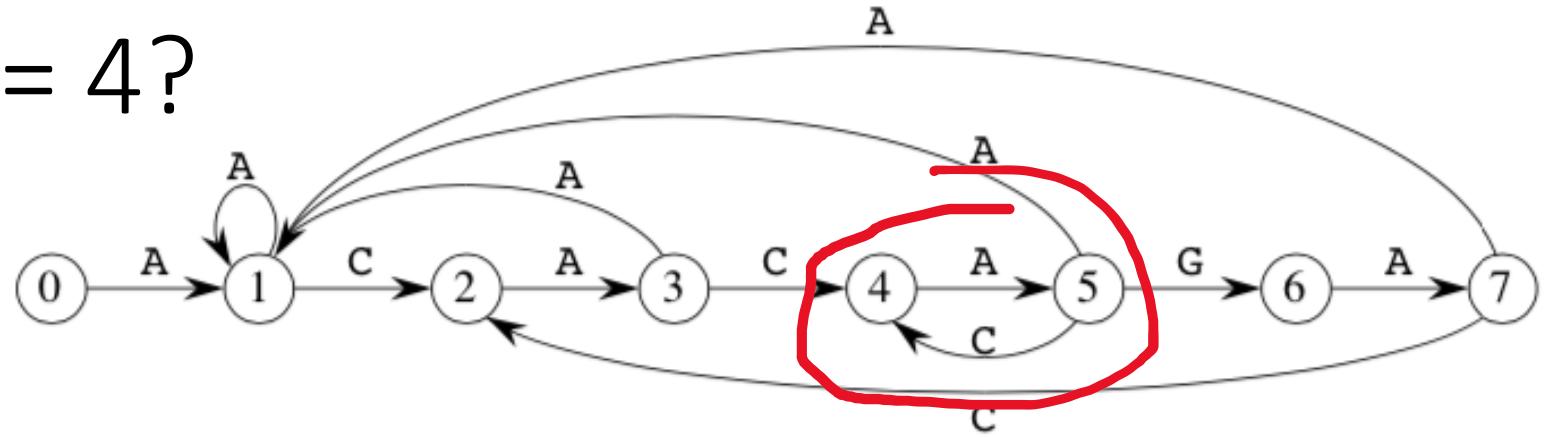
The FA of the pattern ACACAGA:



Why is δ(5,c) = 4?

If the FA gets to state 5 then the 5 most recent characters read are ACACA (see the "spine")
If the next character read is C then it doesn't match so the FA cannot go to state 6. But it does not need to go back to state 0 either because now the 4 most recently read characters are ACAC: first 4 characters of the pattern so the FA moves to state 4

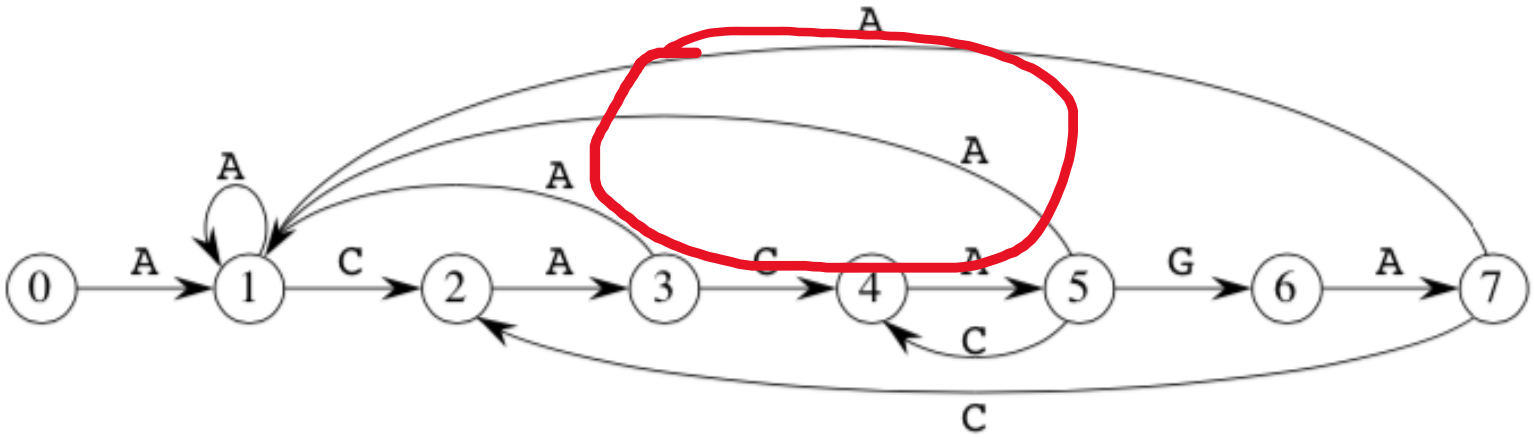
Why is δ(5,c) = 4?



- The **prefix** P_i of a string P is a substring consisting of the first I characters of P
- The **suffix** is a substring consisting of characters from the end
- For a string X and a character a denote concatenating a onto X by Xa
- In state k we most recently read P_k in the text
- We look at the next character a of the text so now we've read P_k a
- How long a prefix of P have we just read:

Find the longest prefix of P that is also a suffix of P_k a. Then δ(k,a) should be the length of the longest prefix

Why is δ(5,c) = 4?



- Take the prefix $P_5 = \texttt{ACACA}$.
- Concatenate `C`, giving `ACACAC`.
- Want the longest prefix of the pattern `ACACAGA` that is also a suffix of `ACACAC`.
- Because the length of `ACACAC` is 6 and the suffix can't be longer than that, start by looking at P_6. Work our way down to shorter and shorter prefixes until we find a prefix that is also a suffix of `ACACAC`.
- P_6 is `ACACAG`. Not a suffix of `ACACAC`.
- P_5 is `ACACA`. Not a suffix of `ACACAC`.
- P_4 is `ACAC`. This is a suffix of `ACACAC`, so stop and set $\delta(5, \texttt{C}) = 4$.

```
COMPUTE-δ(P, Σ, m)
  for q = 0 to m
      for each character a ∈ Σ
          k = min(m + 1, q + 2)
          repeat
              k = k − 1
          until P_k is a suffix of P_q a
          set δ(q, a) = k
  return δ
```

We're guaranteed to stop, because P_0 is the empty string, and the empty string is a suffix of any string. So if we don't find a suffix of $P_k a$ from among $P_k, P_{k-1}, P_{k-2}, \ldots, P_1$, then set $\delta(k, a) = 0$.

Because k is decremented in the **repeat** loop before the first suffix test, its maximum value is $\min(m, q+1)$. If we set $\delta(q,a) = q+1$, then P_{q+1} is a suffix of $P_q a$, and so we have found the next character in the pattern. We cap k at m, since we can't go to a state numbered higher than m.

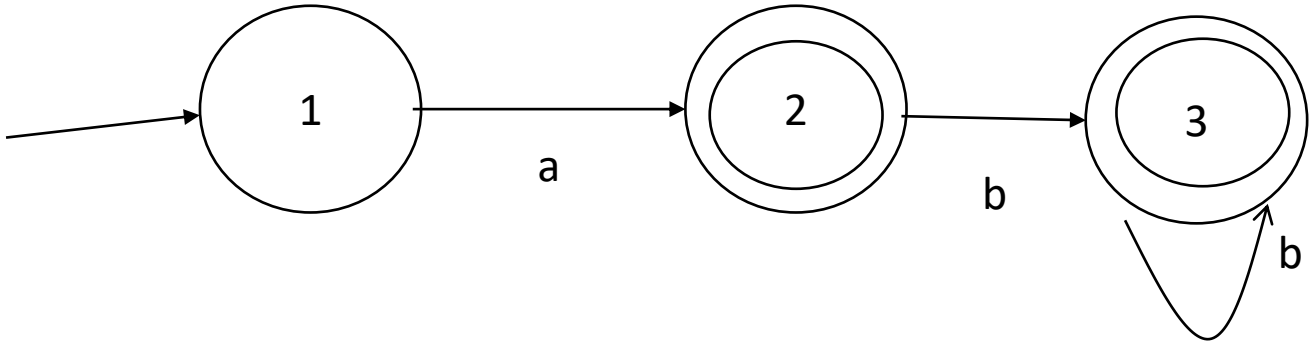
Preprocessing time:

- The outermost **for** loop iterates $m+1$ times.
- The middle loop iterates $|\Sigma|$ times.
- The **repeat** loop iterates at most $m+1$ times.
- Each suffix check in the **until** part examines at most m characters of the pattern (since $k \le m$). Therefore, each suffix check takes $O(m)$ time.
- Total preprocessing time: $O(m^3 |\Sigma|)$.

Therefore, with a finite automaton, we can perform string matching with preprocessing time $O(m^3 |\Sigma|)$ and matching time $\Theta(n)$. It's possible to get the preprocessing time down to $O(m |\Sigma|)$.

DFA and Regular Expressions

- Regular Expressions can be represented with DFA, for example:
- ab* means any string starting with a and then and then 0 or more b



- **Note**: Here the meaning of the wildcard * could be different from regular expressions in some programming languages. Sometimes the regex used in programming languages uses * to represent any number of any character. Here it is formally used to indicate any number of just the preceding character.

Regular expressions

A regular expression (RE) is a notation for specifying a set of strings (a formal language).

An RE is either

- The empty set
- The empty string
- A single character or wildcard symbol
- An RE enclosed in parentheses
- The *concatenation* of two or more REs
- The *union* of two or more REs
- The *closure* of an RE (any number of occurrences)

operation	*example RE*	*matches* (IN *the set*)	*does not match* (NOT *in the set*)
concatenation	aabaab	aabaab	*every other string*
wildcard	.u.u.u.	cumulus jugulum	succubus tumultuous
union	aa | baab	aa baab	*every other string*
closure	ab*a	aa abbba	ab ababa
parentheses	a(a|b)aab	aaaab abaab	*every other string*
	(ab)*a	a abababa	aa abbba

More examples of regular expressions

The notation is surprisingly expressive.

regular expression	*matches*	*does not match*
`.*spb.*` *contains the trigraph* spb	raspberry crispbread	subspace subspecies
`a*	(a*ba*ba*ba*)*` *multiple of three* b's	bbb aaa bbbaababbaa	b bb baabbbaa
`.*0....` *fifth to last digit is* 0	1000234 98701234	111111111 403982772
`.*gcg(cgg|agg)*ctg.*` *fragile X syndrome pattern*	...gcgctg... ...gcgcggctg... ...gcgcggaggctg...	gcgcgg cggcggcggctg gcgcaggctg

Generalized regular expressions

Additional operations further extend the utility of REs.

operation	*example RE*	*matches*	*does not match*
one or more	`a(bc)+de`	abcde abcbcde	ade bcde
character class	`[A-Za-z][a-z]*`	lowercase Capitalized	camelCase 4illegal
exactly j	`[0-9]{5}-[0-9]{4}`	08540-1321 19072-5541	111111111 166-54-1111
between j and k	`a.{2,4}b`	abcb abcbcb	ab aaaaaab
negation	`[^aeiou]{6}`	rhythm	decade
whitespace	`\s`	*any whitespace char (space, tab, newline...)*	*every other character*

Note. These operations are all *shorthand*.
They are very useful but not essential.

RE: `(a|b|c|d|e)(a|b|c|d|e)*`
shorthand: `(a-e)+`

Example of describing a pattern with a generalized RE

A C_2H_2-type zinc finger domain signature is

- C followed by 2, 3, or 4 amino acids, followed by
- C followed by 3 amino acids, followed by
- L, I, V, M, F, Y, W, C, or X followed by 8 amino acids, followed by
- H followed by 3, 4, or 5 amino acids, followed by

Q. Give a generalized RE for all such signatures.

A. C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H

"Wildcard" matches any of the letters
CAVLIMCRKHDENQSTYFWP

Constructs of the standard regular expression and meta characters

Let's get familiar with core constructs of regular expressions and some reserve meta characters that have a special meaning in regular expressions.

Symbol	Meaning	Example
. (dot or period)	Matches any character other than newline.	Matches **#**, **@**, **A**, **f**, **5**, or **.**
***** (asterisk)	***** matches zero or more occurrences of the preceding character or group.	`m*` matches 0 or more occurrences of the letter **m.**
+ (plus)	**+** matches one or more occurrences of the preceding element.	`m+` matches one or more occurrences of the letter **m.**
? (question mark)	**?** means optional match. It is used to match zero or one occurrence of the preceding element. It is also used for lazy matching (which will be covered in the coming chapters).	`nm?` means match **n** or **nm**, as **m** is an **optional** match here.

Lazy and Greedy Matching

- Quantifiers are simple but can be confusing
- When looking for a match the reg ex evaluator does this:

  For every position in the string

  Try to match the pattern at that position.

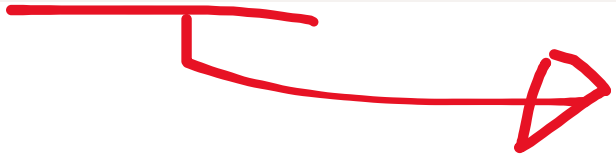
  If there's no match, go to the next position.

- Greedy matching quantifiers, such as **".+"** attempt to match as many instances of a pattern as possible in a string
- Lazy matching using **"?"** (e.g. ".?" or "*?") changes this behavior to match the minimal number of times

```
let regexp = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch" and her "broom"
```

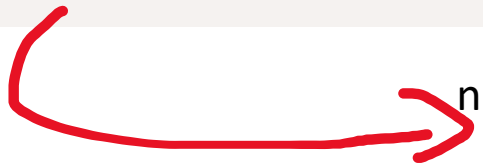


returns the whole string:
"witch" and her "broom"

```
let regexp = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // witch, broom
```



now it returns the two matching strings:
"witch","broom"

Symbol	Meaning	Example
| (pipe)	| means alternation. It is used to match one of the elements separated by |	`m|n|p` means match either the letter `m` or the letter `n` or the letter `p`
^ (cap)	^ is called anchor, that matches start of the line	`^m` matches `m` only when it is the first character of the string that we are testing against the regular expression. Also, note that you do not use ^ in the middle of a regular expression.
$ (dollar)	**$** is called anchor that matches line end.	`m$` matches `m` only at line end.

Symbol	Meaning	Example
\b (backslash followed by the letter b)	Alphabets, numbers, and underscore are considered word characters. **\b** asserts word boundary, which is the position just before and after a word.	`\bjava\b` matches the word, `java` . So, it will not match `javascript` since the word, `javascript`, will fail to assert `\b` after `java` in the regex.
\B (backslash followed by uppercase B)	**\B** asserts true where **\b** doesn't, that is, between two word characters.	For the input text, *abc*, `\B` will be asserted at two places: 1. Between *a* and *b*. 2. Between *b* and *c*.
(...) a sub-pattern inside round parentheses	This is for grouping a part of text that can be used to capture a certain substring or for setting precedence.	`m(ab)*t` matches `m`, followed by zero or more occurrences of the substring, **ab**, followed by **t**.
{min,max}	A quantifier range to match the preceding element between the minimum and the maximum number.	`mp{2,4}` matches `m` followed **2** to **4** occurrences of the letter `p`.

Symbol	Meaning	Example
[...]	This is called a character class.	`[A-Z]` matches any uppercase English alphabet.
\d (backslash followed by the letter d)	This will match any digit.	`\d` matches any digit in the 0-9 range.
\D (backslash followed by uppercase D)	This matches any character that is not a digit.	`\D` matches `a`, `$`, or `_`.
\s (backslash followed by the letter s)	Matches any whitespace, including tab, space, or newline.	`\s` matches `[ \t\n]`.
\S (backslash followed by uppercase S)	Matches any non-whitespace.	`\S` matches the opposite of `\s`

Symbol	Meaning	Example
\w (backslash followed by the letter w)	Matches any word character that means all alphanumeric characters or underscore.	`\w` will match `[a-zA-Z0-9_]`, so it will match any of these strings: "*abc*", "*a123*", or "*pq_12_ABC*"
\W (backslash followed by the letter W)	Matches any non-word character, including whitespaces. In regex, any character that is not matched by \w can be matched using \W.	It will match any of these strings: "*+/=*", "*$*", or " *!~*"

Some basic regular expression examples

`ab*c`	This will match `a`, followed by zero or more `b`, followed by `c`.
`ab+c`	This will match `a` followed by one or more `b`, followed by `c`.
`ab?c`	This will match `a` followed by zero or one `b`, followed by `c`. Thus, it will match both `abc` or `ac`.
`^abc$`	This will match `abc` in a line, and the line must not have anything other than the string `abc` due to the use of the start and end anchors on either side of the regex.
`a(bc)*z`	This will match `a`, followed by zero or more occurrences of the string `bc`, followed by `z`. Thus, it will match the following strings: `az`, `abcz`, `abcbcz`, `abcbcbcz`, and so on.

`a(bc)*z`

This will match `a`, followed by zero or more occurrences of the string `bc`, followed by `z`. Thus, it will match the following strings: `az`, `abcz`, `abcbcz`, `abcbcbcz`, and so on.

`ab{1,3}c`

This will match `a`, followed by one to three occurrences of `b`, followed by `c`. Thus, it will match following strings: `abc`, `abbc`, and `abbbc`.

`red|blue`

This will match either the string `red` or the string `blue`.

`\b(cat|dog)\b`

This will match either the string `cat` or the string `dog`, ensuring both `cat` and `dog` must be complete words; thus, it will **fail** the match if the input is `cats` or `dogs`.

`[0-9]`

This is a character class with a character range. The preceding example will match a digit between `0` and `9`.

| `[a-zA-Z0-9]`

This is a character class with a character range. The preceding example will match any alpha-numeric character.

| `^\d+$`

This regex will match an input containing only one or more digits.

| `^\d{4,8}$`

This regex will allow an input containing four to eight digits only. For example, `1234`, `12345`, `123456`, and `12345678` are valid inputs.

| `^\d\D\d$`

This regex not only allows only one digit at the start and end but also enforces that between these two digits there must be one non-digit character. For example, `1-5`, `3:8`, `8X2`, and so on are valid inputs.

| `^\d+\.\d+$`

This regex matches a floating point number. For example, `1.23`, `1548.567`, and `7876554.344` are valid inputs.

| `.+`

This matches any character one or more times. For example, `qwqewe`, `12233`, or `f5^h_=!bg` are all valid inputs:

| `^\w+\s+\w+$`

This matches a word, followed by one or more whitespaces, followed by another word in an input. For example, `hello word`, `John Smith`, and `United Kingdom` will be matched using this regex.

Another real world application

Database of protein domains, families and functional sites

SARS-CoV-2 relevant PROSITE motifs

PROSITE consists of documentation entries describing protein domains, families and functional sites as well as associated patterns and profiles to identify them [More... / References / Commercial users].
PROSITE is complemented by ProRule , a collection of rules based on profiles and patterns, which increases the discriminatory power of profiles and patterns by providing additional information about functionally and/or structurally critical amino acids [More...].

Release 2020_02 of 22-Apr-2020 contains 1858 documentation entries, 1311 patterns, 1277 profiles and 1301 ProRule.

Search

e.g. PDOC00022, PS50089, SH3, zinc finger

Search

Type a regular expression here

Browse

- by documentation entry
- by ProRule description
- by taxonomic scope
- by number of positive hits

Quick Scan mode of ScanProsite

Other tools

Advanced RegEx

The effect of eager matching on regular expression alternation

This regular expression engine behavior may return unexpected matches in alternation if alternations are not ordered carefully in the regex pattern.

Take an example of this regex pattern, which matches the strings `white` or `whitewash`:

```
|white|whitewash
```

While applying this regex against an input of *whitewash*, the regex engine finds that the first alternative `white` matches the *white* substring of the input string *whitewash*, hence, the regex engine stops proceeding further and returns the match as `white`.

Note that our regex pattern has a better second alternative as `whitewash`, but due to the regex engine's eagerness to complete and return the match, the first alternative is returned as a match and the second alternative is ignored.

However, consider swapping the positions of the third and fourth alternatives in our regex pattern to make it as follows:

```
|whitewash|white
```

If we apply this against the same input, *whitewash*, then the regex engine correctly returns the match as `whitewash`.

Basic quantifiers

The following table lists all the quantifiers available in Java regular expressions:

Quantifier	Meaning
m*	Match **m** zero or more times
m+	Match **m** one or more times
m?	Match **m** one or zero times (also called an optional match)
m{X}	Match **m** exactly *X* times
m{X,}	Match **m** *X* or more times
m{X,Y}	Match **m** at least *X* and at most *Y* times

*In all the aforementioned cases, **m** can be a single character or a group of characters. We will discuss grouping in more detail later.*

Examples using quantifiers

Let's look at few examples to understand these basic quantifiers better.

Which regex pattern should be used to match a two-digit year or a four-digit year?

```
|\d{2}|\d{4}
```

Which regex pattern should be used to match a signed decimal number? The pattern should also match a signed integer number:

```
|^[+-]?\d*\.?\d+$
```

Here is the breakup of the preceding regex pattern:

- The `^` and `$` symbols are the start/end anchors
- The `[+-]?` pattern makes either the `+` sign or the `-` sign (optional because of `?`) at the start
- The `\d*` pattern matches zero or more digits
- The `\.?` pattern matches an optional dot (`.`) literally
- The `\d+` pattern matches one or more digits

The preceding regex will match all of these inputs:

- .45
- 123789
- 5
- 123.45
- +67.66
- -987.34

```
|^[+-]?\d*\.?\d+$
```

What would be the regex to match a number that is at least `10` but not more than `9999`?

```
|^\d{2,4}$
```

Since we have a minimum of two digits, `10` is the smallest match, whereas the maximum number of digits allowed is four, and hence, `9999` is the highest match.

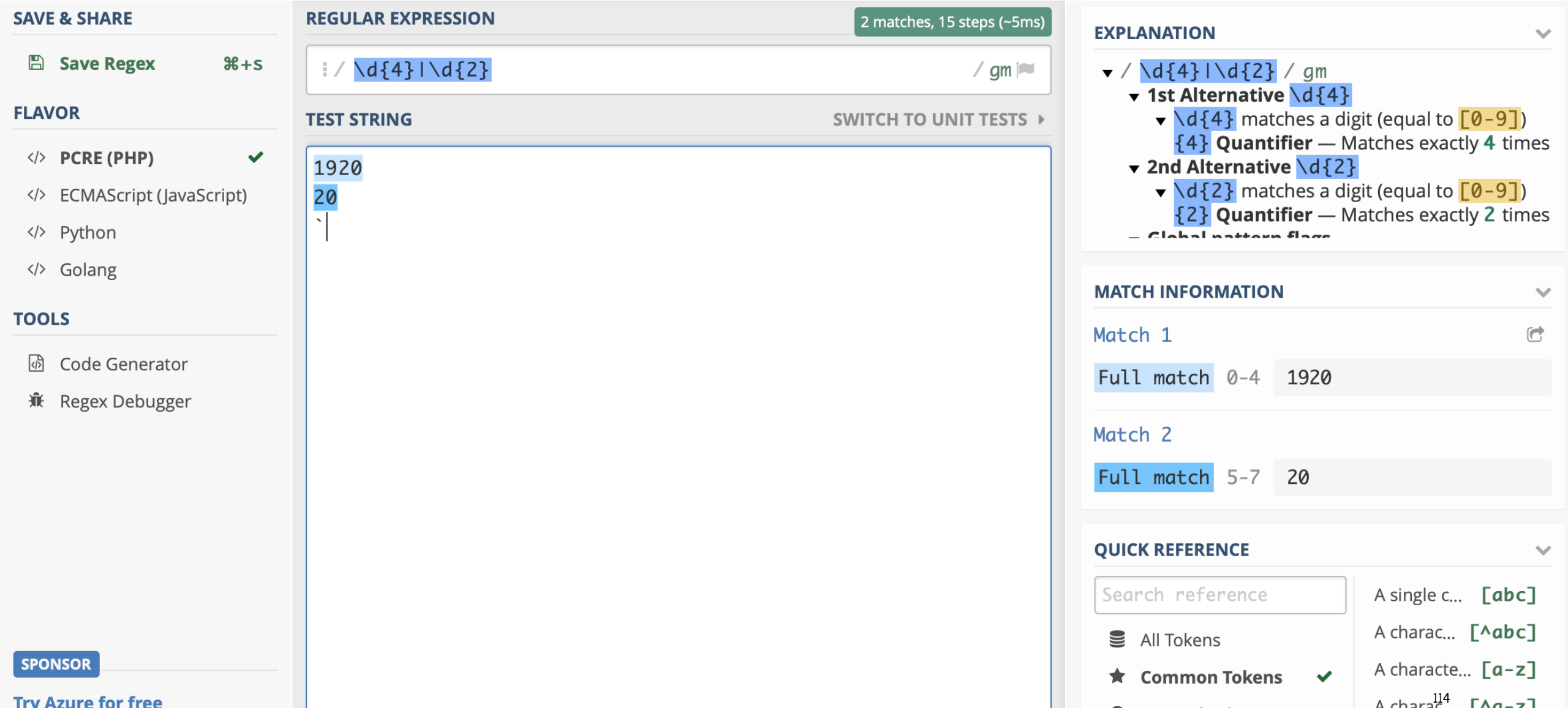
What is the regex for an input that has seven digits and that can have `+` or `-` at the start?

```
|^[+-]?\d{7}$
```

The `[+-]?` pattern makes it an optional match at the start before we match the seven digits using `\d{7}`.

> *The preceding regex can also be written as `^[+-]?[0-9]{7}$`, as `\d` is a shorthand property to match `[0-9]`*

If confusing try it here: https://regex101.com



Possessive quantifiers

Possessive quantifiers are quantifiers that are greedy when matching text like greedy quantifiers do. Both greedy and possessive quantifiers try to match as many characters as possible. The important difference, however, is that the possessive quantifiers do not backtrack (go back) unlike greedy quantifiers; therefore, it is possible that the regex match fails if the possessive quantifiers go too far.

This table shows all the three types of quantifiers, side by side:

Greedy Quantifier	Lazy Quantifier	Possessive Quantifier
m*	m*?	m*+
m+	m+?	m++
m?	m??	m?+
m{X}	m{X}?	m{X}+
m{X,}	m{X,}?	m{X,}+
m{X,Y}	m{X,Y}?	m{X,Y}+

Let's take an example input string `a1b5`, and see the behavior of the greedy, lazy, and possessive quantifiers.

If we apply a regex using the greedy quantifier, `\w+\d`, then it will match `a1b` (the longest match before backtracking starts) using `\w+`, and `5` will be matched using `\d`; thus, the full match will be `a1b5`.

Now, if we apply a regex using the non-greedy quantifier, `\w+?\d`, then it will match `a` (the shortest match before expanding starts) using `\w+?`, and then the adjacent digit `1` will be matched using `\d`. Thus, the first full match will be `a1`. If we let the regex execute again, then it will find another match, `b5`.

Finally, if we apply a regex using the possessive quantifier, `\w++\d`, then it will match all the characters `a1b5` (the longest possible match without giving back) using `\w++` . Due to this, `\d` remains unmatched, and hence the regex fails to find any match.

Let's take another example. The requirement is to match a string that starts with lowercase English alphabets or hyphen. The string can have any character after the alphabets/hyphens, except a colon. There can be any number of any characters of any length after the colon until the end.

An example of a valid input is `as-df999` and that of an invalid input is `asdf-:123`.

Now, let's try solving this regex problem using a greedy quantifier regex:

```
^[a-z-]+[^:].*$
```

Unfortunately, this is not the right regex pattern because this regex will match both the aforementioned valid and invalid inputs. This is because of the backtracking behavior of the regex engine in greedy quantifiers. The `[a-z-]+` pattern will find the longest possible match in the form of `asdf-`, but due to the negated character class pattern `[^:]` , the regex engine will backtrack one position to `asdf` and will match the next *hyphen* for `[^:]`. All the remaining text, that is, `:123`, will be matched using `.*`.

An example of a valid input is `S as-df999` and that of an invalid input is `asdf-:123`.

> Let's try to solve this regex problem using the following possessive quantifier regex:
>
> ```
> |^[a-z-]++[^:].*$
> ```
>
> This regex pattern will still match our valid input, but it will fail to match an invalid input because there is no backtracking in possessive quantifiers; hence, the regex engine will not go back any position after matching `asdf-` in the second example string. Since the next character is a colon and our regex sub-pattern is `[^:]`, the regex engine will stop matching and correctly declare our invalid input a failed match.
>
> Possessive quantifiers are good for the performance of the underlying regex engine because the engine does not have to keep any backtracking information in memory. The performance increase is even more when a regex fails to match because possessive quantifiers fail faster.

Another example of describing a pattern with a generalized RE

An e-mail address is

- A sequence of letters, followed by
- the character "@", followed by
- the character "." , followed by a nonempty sequence of lowercase letters, followed by
- [any number of occurrences of the previous pattern]
- "edu" or "com" (others omitted for brevity).

Q. Give a generalized RE for e-mail addresses.

A. `[a-z]+@([a-z]+\.)+(edu|com)`

Exercise. Extend to handle `rs123@princeton.edu`, more suffixes such as `.org`, and any other extensions you can think of.

Next. Determining whether a given string matches a given RE.

Regular Expressions with Java

Week 14 – Presentation 4

REs in Java

Java's String class implements GREP.

public class String	
...	
boolean matches(String re)	*does this string match the given RE?*
...	

```
String re = "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H";
String zincFinger = "CAASCGGPYACGGAAGYHAGAH";
boolean test = zincFinger.matches(re);
```

true!

Java RE client example: Validation

```
public class Validate
{
   public static void main(String[] args)
   {
      String re = args[0];
      while (!StdIn.isEmpty())
      {
         String input = StdIn.readString();
         StdOut.println(input.matches(re));
      }
   }
}
```

Does a given string match a given RE?

- Take RE from command line.
- Take strings from StdIn.

need quotes to "escape" the shell

```
% java Validate "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H"
CAASCGGPYACGGAAGYHAGAH
true
CAASCGGPYACGGAAGYHGAH
false

% java Validate "[$_A-Za-z][$_A-Za-z0-9]*"
ident123
true
123ident
false

% java Validate "[a-z]+@([a-z]+\.)+(edu|com)"
wayne@cs.princeton.edu
true
eve@airport
false
```

C_2H_2 type zinc finger domain

legal Java identifier

valid email address (simplified)

Applications

- Scientific research.
- Compilers and interpreters.
- Internet commerce.
- ...

Beyond matching

Java's `String` class contains other useful RE-related methods.

- RE search and replace
- RE delimited parsing

`public class String`	
`...`	
`String replaceAll(String re, String to)`	*replace all occurrences of substrings matching RE with* `to`
`String[] split(String re)`	*split the string around matches of the given RE*
`...`	

Tricky notation (typical in string processing): \ signals "special character" so "\\" means "\" and "\\s" means "\s"

Examples using the RE `"\\s+"` (matches one or more whitespace characters).

Replace each sequence of at least one whitespace character with a single space.

```
String s = StdIn.readAll();
s = s.replaceAll("\\s+", " ");
```

Create an array of the words in `StdIn` (basis for `StdIn.readAllStrings()` method)

```
String s = StdIn.readAll();
String[] words = s.split("\\s+");
```

Java String API for regular expressions' evaluation

Method Signature	Purpose
boolean matches(String regex)	Matches the given regular expression against the string that the method is invoked on and returns true/false, indicating whether the match is successful (true) or not (false).
String `replaceAll`(String regex, String replacement)	Replaces each substring of the subject string that matches the given regular expression with the replacement string and returns the new string with the replaced content.
String `replaceFirst`(String regex, String replacement)	This method does the same as the previous one with the exception that it replaces only the first substring of the subject string that matches the given regular expression with the replacement string and returns the new string with the replaced content.
`String[]` split(String regex)	Splits the subject string using the given regular expression into an array of substrings (example given ahead).
`String[]` split(String regex, int limit)	This overloaded method does the same as the previous one but there is an additional second parameter. The `limit` parameter controls the number of times regular expressions are applied for splitting.

Way beyond matching

java.util.regex

Java's Pattern and Matcher classes give fine control over the GREP implementation.

public class Pattern	
...	
static Pattern compile(String re)	*parse the* re *to construct a* Pattern
Matcher matcher(String input)	*create a* Matcher *that can find substrings matching the pattern in the given input string*
...	

Why not a constructor? Good question.

public class Matcher	
...	
boolean find()	*set internal variable* match *to the next substring that matches the RE in the input. If none, return* false*, else return* true
String group()	*return* match
String group(int k)	*return the kth group (identified by parens within RE) in* match
...	

[A sophisticated interface designed for pros, but very useful for everyone.]

Java pattern matcher client example: Harvester

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
   public static void main(String[] args)
   {
      String re      = args[0];
      In in          = new In(args[1]);
      String input   = in.readAll();
      Pattern pattern = Pattern.compile(re);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         StdOut.println(matcher.group());

   }
}
```

Harvest information from input stream

- Take RE from command line.
- Take input from file or web page.
- Print all substrings matching RE.

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg

% java Harvester "[a-z]+@([a-z]+\.)+(edu|com)" http://www.cs.princeton.edu/people/faculty
...
rs@cs.princeton.edu
...
wayne@cs.princeton.edu
...
```

harvest patterns from DNA

harvest email addresses from web for spam campaign.
(no email addresses on that site any more)

Using regular expressions in Java Scanner API

A scanner is a utility class used for parsing the input text and breaking the input into tokens of various types, such as Boolean, int, float, double, long, and so on. It generates tokens of various types using regular expression-based delimiters. The default delimiter is a whitespace. Using the Scanner API, we can generate tokens of all the primitive types in addition to string tokens.

The `String`, `Pattern`, and `Matcher` classes are able to parse the input and generate tokens of the `String` type only, but the `Scanner` class is very useful for checking and generating tokens of different types from the input source. The `Scanner` instance can be constructed using the `File`, `InputStream`, `Path`, `Readable`, `ReadableByteChannel`, and `String` arguments.

Method Signature	Purpose
Scanner useDelimiter(String pattern)	Sets this scanner's delimiter regex pattern to a String regex argument.
Scanner useDelimiter(Pattern pattern)	This method is almost the same as the previous one but gets a `Pattern` as an argument instead of a `String`. This means that we can pass a regular expression already compiled. If we are forced to use the version with the `String` argument, the scanner would compile the string to a `Pattern` object even if we have already executed that compilation in other parts of the code. We will discuss the `Pattern` and `Matcher` class in the next chapter.
Pattern delimiter()	Returns the pattern being used by this scanner to match delimiters.
MatchResult match()	Returns the match result of the latest scan operation performed by this scanner.

Method Signature	Purpose
boolean hasNext(String pattern)	Returns `true` if the next token matches the pattern constructed from the specified string.
boolean hasNext(Pattern pattern)	This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`.
String next(String pattern)	Returns the next token if it matches the pattern constructed from the specified string.
String next(Pattern pattern)	This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`.
String findInLine(String pattern)	Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

Method Signature	Purpose
String findInLine(Pattern pattern)	This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`.
Scanner skip(String pattern)	Skips the input that matches a pattern constructed from the specified string, ignoring delimiters.
Scanner skip(Pattern pattern)	This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`.
String findWithinHorizon(String pattern, int horizon)	Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String findWithinHorizon(Pattern pattern, int horizon)	This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`.

Package java.util.regex

Java Regular Expression API - Pattern and Matcher Classes

- The `MatchResult` interface
- Using `Pattern` class
- Using `Matcher` class
- Various methods of `Pattern` and `Matcher` classes and how to use them for solving problems involving regular expressions

The MatchResult interface

`MatchResult` is an interface for representing the result of a match operation. This interface is implemented by the `Matcher` class.

Method Name	Description
int start()	Returns the start index of the match in the input
int start(int group)	Returns the start index of the specified capturing group
int end()	Returns the offset after the last character matched
int end(int group)	Returns the offset after the last character of the subsequence captured by the given group during this match
String group()	Returns the input substring matched by the previous match
String group(int group)	Returns the input subsequence captured by the given group during the previous match operation
int groupCount()	Returns the number of capturing groups in this match result's pattern

Let's take an example to understand this interface better.

Suppose, the input string is a web server response line from HTTP response headers:

```
HTTP/1.1 302 Found
```

Our regex pattern to parse this line is as follows:

```
HTTP/1\.[01] (\d+) [a-zA-Z]+
```

Note that there is only one captured group that captures integer status code.

Let's look at this code listing to understand the various methods of the `MatchResult` interface better:

```java
package example.regex;

import java.util.regex.*;

public class MatchResultExample
{
  public static void main(String[] args)
  {
    final String re = "HTTP/1\\.[01] (\\d+) [a-zA-Z]+";
    final String str = "HTTP/1.1 302 Found";

    final Pattern p = Pattern.compile(re);
    Matcher m = p.matcher(str);

    if (m.matches())
    {
      MatchResult mr = m.toMatchResult();

      // print count of capturing groups
      System.out.println("groupCount(): " + mr.groupCount());

      // print complete matched text
      System.out.println("group(): " + mr.group());

      // print start position of matched text
      System.out.println("start(): " + mr.start());

      // print end position of matched text
      System.out.println("end(): " + mr.end());

      // print 1st captured group
      System.out.println("group(1): " + mr.group(1));

      // print 1st captured group's start position
      System.out.println("start(1): " + mr.start(1));

      // print 1st captured group's end position
      System.out.println("end(1): " + mr.end(1));
    }
  }
}
```

```
groupCount(): 1
group(): HTTP/1.1 302 Found
start(): 0
end(): 18
group(1): 302
start(1): 9
end(1): 12
```

See the Docs…

Class Summary

Class	Description
Matcher	An engine that performs match operations on a **character sequence** by interpreting a **Pattern**.
Pattern	A compiled representation of a regular expression.

https://docs.oracle.com/javase/8/docs/api/index.html?java/util/regex/package-summary.html

Convert Regex to Predicate

Use `Pattern.compile().asPredicate()` method to get a predicate from compiled regular expression.

This predicate can be used with lambda streams to apply it on each token in a stream.

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b
```

What does this match?

More Case Studies

Matching special characters using backslash to escape

```java
import java.util.*;
import java.util.regex.*;

public class PatternQuoteExample {
    public static void main (String[] args) {
        String input = "Math operators: +-*/. ";
        boolean result;

        String quoted = Pattern.quote("+-*/.");
        System.out.println(quoted);

        // regex using standard escaping
        result = input.matches(".*\\s+\\+-\\*/\\.\\s+.*");
        System.out.println(result);

        // regex Using Pattern.quote around our search string
        result = input.matches(".*\\s+" + quoted + "\\s+.*");
        System.out.println(result);

        // regex Using \Q and \E around our search string
        result = input.matches(".*\\s+\\Q+-*/.\\E\\s+.*");
        System.out.println(result);
    }
}
```

```
\Q+-*/.\E
true
true
true
```

```java
import java.util.*;
import java.util.regex.*;
public class PatternSplitExample {
    public static void main (String[] args) {
        final String input = "value1||value2||value3";
        final Pattern p = Pattern.compile( Pattern.quote( "||" ) );

        // call split and print each element from generated array
        // using stream API
        Arrays.stream( p.split(input) )  // p.splitAsStream(input)
              .forEach( System.out::println );
    }
}
```

```
H:\work\CS209A_19S\notes08>javac PatternSplitExample.java

H:\work\CS209A_19S\notes08>java PatternSplitExample
value1
value2
value3
```

Splitting using regular expressions instead of string operations

```java
import java.util.List;
import java.util.stream.*;
import java.util.regex.*;

public class AsPredicateExample {
   public static void main (String[] args) {
      final String[] monthsArr =
         { "10", "0", "05", "09", "12", "15", "00", "-1", "100" };
      final Pattern validMonthPattern =
         Pattern.compile( "^(?:0?[1-9]|1[0-2])$" );
      List<String> filteredMonths =
         Stream.of( monthsArr )
               .filter( validMonthPattern.asPredicate() )
               .collect( Collectors.toList() );
      System.out.println( filteredMonths );
   }
}
```

```
H:\work\CS209A_19S\notes08>javac AsPredicateExample.java

H:\work\CS209A_19S\notes08>java AsPredicateExample
[10, 05, 09, 12]
```

Making a predicate and using stream processing supported by regex

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class RegexPredicateExample {
    public static void main(String[] args) {
        // Compile regex as predicate
        Predicate<String> emailFilter =
            Pattern.compile("^(.+)@example.com$")
                   .asPredicate();

        // Input list
        List<String> emails = Arrays.asList(
            "alex@example.com", "bob@yahoo.com",
            "cat@google.com",   "david@example.com"
        );

        // Apply predicate filter
        List<String> desiredEmails =
            emails.stream()
                  .filter(emailFilter)
                  .collect(Collectors.<String>toList());

        // Now perform desired operation
        desiredEmails.forEach(System.out::println);
    }
}
```

Making a predicate and using stream processing supported by regex applied to filter emails by domain

```
alex@example.com
david@example.com
```

```java
import java.util.regex.*;

public class MatcherMatchesExample {
   public static void main (String[] args) {
      final Pattern pattern1 = Pattern.compile( "mastering" );
      final Pattern pattern2 = Pattern.compile( "mastering.*" );
      final Pattern pattern3 = Pattern.compile( "regular.*" );

      String input = "mastering regular expressions";
      Matcher matcher = pattern1.matcher(input);
      System.out.printf( "[%s] => [%s]: %s%n",
         input, matcher.pattern(), matcher.matches());

      // update the matcher pattern with a new pattern
      matcher.usePattern(pattern2);
      System.out.printf( "[%s] => [%s]: %s%n",
         input, matcher.pattern(), matcher.matches());

      // update the matcher pattern with a new pattern
      matcher.usePattern(pattern3);
      System.out.printf( "[%s] => [%s]: %s%n",
         input, matcher.pattern(), matcher.matches());
   }
}
```

```
H:\work\CS209A_19S\notes08>java MatcherMatchesExample
[mastering regular expressions] => [mastering]: false
[mastering regular expressions] => [mastering.*]: true
[mastering regular expressions] => [regular.*]: false
```

Using matching (we set a different pattern in java regex matcher class)

Each pattern may or may not match the string given as input (“mastering regular expressions”)

```java
import java.util.regex.*;
public class MatcherFindExample {
   public static void main (String[] args) {
      final String input =
         "some text <value1> anything <value2><value3> here";

      /* Part 1 */
      final Pattern pattern = Pattern.compile( "<([^<>]*)>" );
      Matcher matcher = pattern.matcher(input);
      while (matcher.find()) {
         System.out.printf( "[%d] => [%s]%n",
            matcher.groupCount(), matcher.group(1) );
      }

      /* Part 2 */
      // now use similar pattern but use a named group and reset the
      // matcher
      matcher.usePattern( Pattern.compile( "<(?<name>[^<>]*)>" ) );
      matcher.reset();
      while (matcher.find()) {
         System.out.printf( "[%d] => [%s]%n",
            matcher.groupCount(), matcher.group("name"));
      }
   }
}
```

```
[1] => [value1]
[1] => [value2]
[1] => [value3]
[1] => [value1]
[1] => [value2]
[1] => [value3]
```

A template to match some elements of a string delineated by angle brackets.

The substrings that are extracted/matched are able to be obtained from the matcher group (they are numbered from 1 to the number of matches).

Part 2 shows the use of a named group instead of a numbered group (easier to read).

```
H:\work\CS209A_19S\notes08>java MatcherAppendExample
<v1=n1 v2=n2 v3=n3> n1=v1 n2=v2 abc=123 <pq=v abc=id> v=pq
```

```java
import java.util.regex.*;

public class MatcherAppendExample {
   public static void main (String[] args) {
      final String input =
         "<n1=v1 n2=v2 n3=v3> n1=v1 n2=v2 abc=123 <v=pq id=abc> v=pq";

      // pattern1 to find all matches between < and >
      final Pattern pattern = Pattern.compile( "<[^>]+>" );

      // pattern1 to find each name=value pair
      final Pattern pairPattern = Pattern.compile( "(\\w+)=(\\w+)" );
      Matcher enclosedPairs = pattern.matcher(input);

      StringBuffer sbuf = new StringBuffer();
      // call find in a loop and call appendReplacement for each match
      while (enclosedPairs.find()) {
         Matcher pairMatcher = pairPattern.matcher( enclosedPairs.group());
         // replace name=value with value=name in each match
         enclosedPairs.appendReplacement(
            sbuf, pairMatcher.replaceAll( "$2=$1" )
         );
      }
      // appendTail to append remaining character to buffer
      enclosedPairs.appendTail( sbuf );
      System.out.println( sbuf );
   }
}
```

Finding multiple matches with a loop (we don't have to know in advance how many matches there will be).

```
java.net.URL  url = new URL( sURL );
```

```
106     /**
107      * Initializes an input stream from a URL.
108      *
109      * @param  url the URL
110      * @throws IllegalArgumentException if cannot open {@code url}
111      * @throws IllegalArgumentException if {@code url} is {@code null}
112      */
113     public In(URL url) {
114         if (url == null) throw new IllegalArgumentException("url argument is null");
115         try {
116             URLConnection site = url.openConnection();
117             InputStream is     = site.getInputStream();
118             scanner            = new Scanner(new BufferedInputStream(is), CHARSET_NAME);
119             scanner.useLocale(LOCALE);
120         }
121         catch (IOException ioe) {
122             throw new IllegalArgumentException("Could not open " + url, ioe);
123         }
124     }
```

Regex are widely used in network programming.

Sample Exam Question:

`Huge` Integers

Huge Integers

In Java a long can have a value that is at most $2^{63}-1$:

$$9,223,372,036,854,775,807$$

Yes its big, but might not be big enough for everyone…

Multiples of 103 (polynomial approach of the problem)…

123,456,789,012,345,678,901 can be written as:

123 x 1000^6
+ 456 x 1000^5
+ 789 x 1000^4
+ 12 x 1000^3
+ 345 x 1000^2
+ 678 x 1000^1
+ 901 x 1000^0

Question – part 1

Of the interfaces available in Java collections (List, Queue/Dequeue, Set), to which we can add the Map interface, which one seems to you the most appropriate to store HugeInteger values?

Answer: Which interface??

One value → ~~Map~~

Same number can appear several times → ~~Set~~

A List or Queue is better

using pseudocode…

Question – part 2 Write (pseudo) code for the two following constructors.

HugeInteger(long intValue)

HugeInteger(String strValue)

The string can contain commas to separate thousands or not. Define exceptions and create specific exceptions if needed, or you may throw any of the following existing exceptions if appropriate:

ArithmeticException ArrayIndexOutOfBoundsException
IllegalArgumentException IndexOutOfBoundsException
NegativeArraySizeException NullPointerException NumberFormatException
StringIndexOutOfBounds UnsupportedOperationException

HugeInteger(long intValue)

```
public HugeInteger(long intvalue)
    List list <- []
    do:
        append (intvalue % 1000) to the list
        intvalue = intvalue / 1000

    while ( intvalue > 1)
```

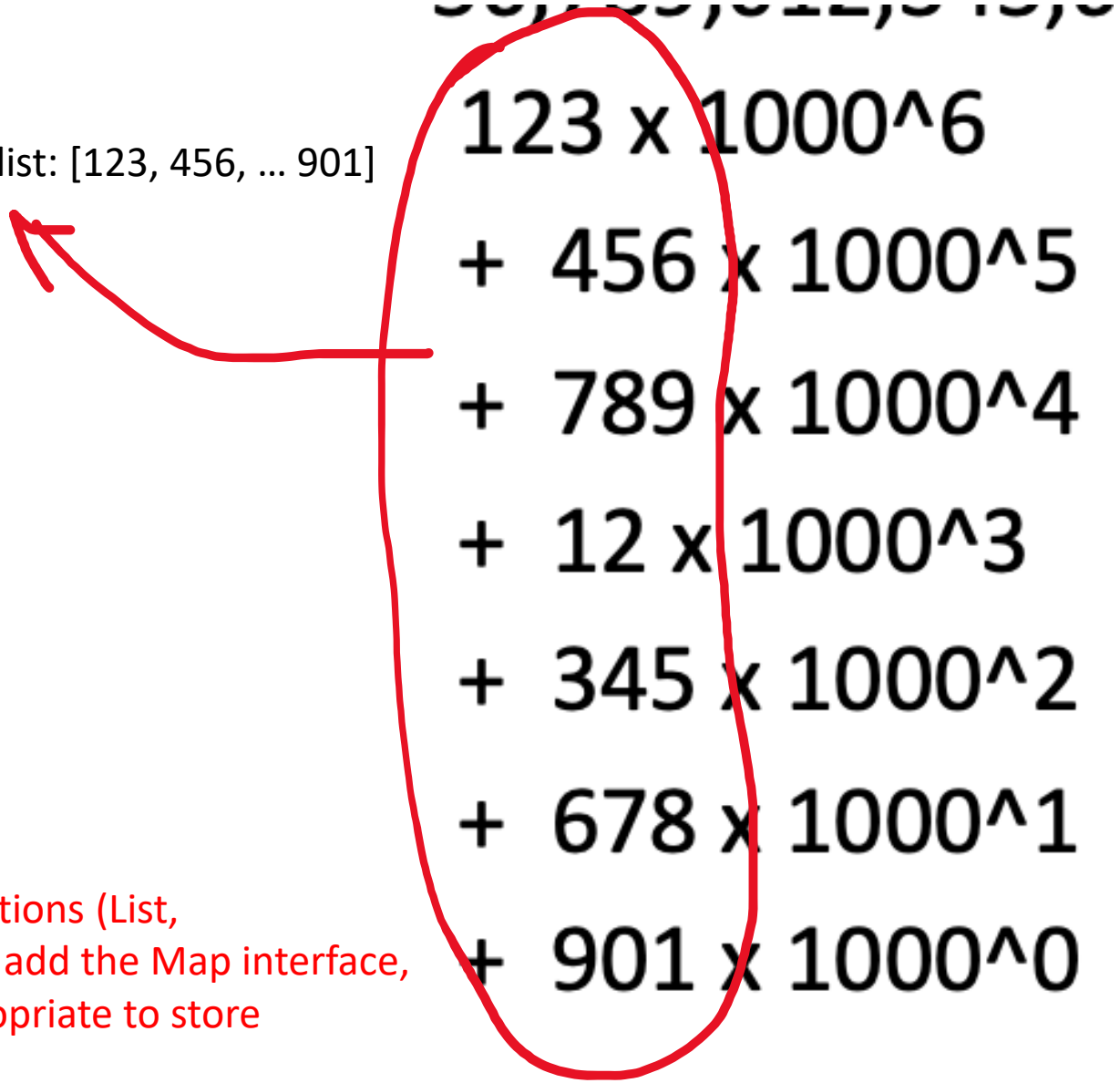
HugeInteger(String strValue)

```
public HugeInteger(String strValue) throws NumberFormatException {
        String str = strValue.replace(",", "");
        while length of strValue > 3 {
                Take the 3 last characters
                Convert to integer using Integer.ParseInt
                add value to list
                set strValue to substring that excludes the last 3 characters
        }

        if length(strValue) > 0 {
                Convert to integer
                add value to list
        }
```

Could also use regular expressions to split/ tokenize the string on ","…

to the list: [123, 456, … 901]



Of the interfaces available in Java collections (List, Queue/Dequeue, Set), to which we can add the Map interface, which one seems to you the most appropriate to store HugeInteger values?

which one seems to you the most appropriate to store HugeInteger values?

- One value -> ~~Map~~
- Same number several times? -> ~~Set~~
- A list or queue perhaps?

Question – part 3

Define an add method to add two HugeInteger objects. Please note that with regular addition the variables that are added are left unmodified (if you have two integer values a and b, the result a + b leaves both a and b unchanged).

What would you prefer – A separate class or a method in the HugeInteger objects? **Justify you preference...**.

Adding 2 HugeInteger objects

No reason to make one of the two objects we are adding more important than the other (would be different with something that would implement a kind of += operation).

A class method makes sense here.

- Give the pseudo code for the method

Pseudo-code

```
HugeInteger result = new HugeInteger();
                                        // Default constructor needed for this

set min to the minimum size of the lists in h1 and h2
set max to the maximum size of the lists in h1 and h2
int sum;
int val;
int carry = 0;
for (int i = 0; i < min; i++) {
        sum = h1.list.get(i) + h2.list.get(i) + carry;
        carry = sum / 1000;
        result.list.add(sum % 1000);
}
```

```
for (int i = min; i < max; i++) {
        if   i >= h1.list.size()   {
                val = h2.list.get(i);
        } else {
                val = h1.list.get(i);
        }
        sum = carry + val;
        carry = sum / 1000;
        result.list.add(sum % 1000);
}
if (carry > 0) {
        result.list.add(carry);
}
return result;
```

Question – part 4

Use the previously defined method to write the pseudo code for a method that adds a long to a HugeInteger. You can assume it exists even if you didn't manage to write it.

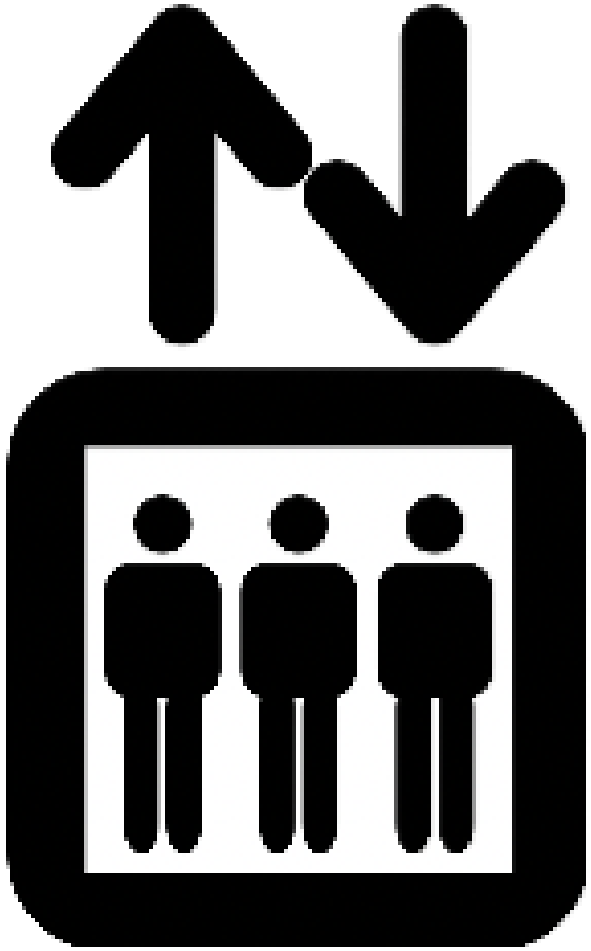
Adding a long to a HugeInteger

```
static HugeInteger add(HugeInteger h, long longVal) {
        return add(h, new HugeInteger(longVal));
}
```

Should also be overloaded with

add(long longVal, HugeInteger h)

to ensure commutativity (ie a*b = b*a).



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.

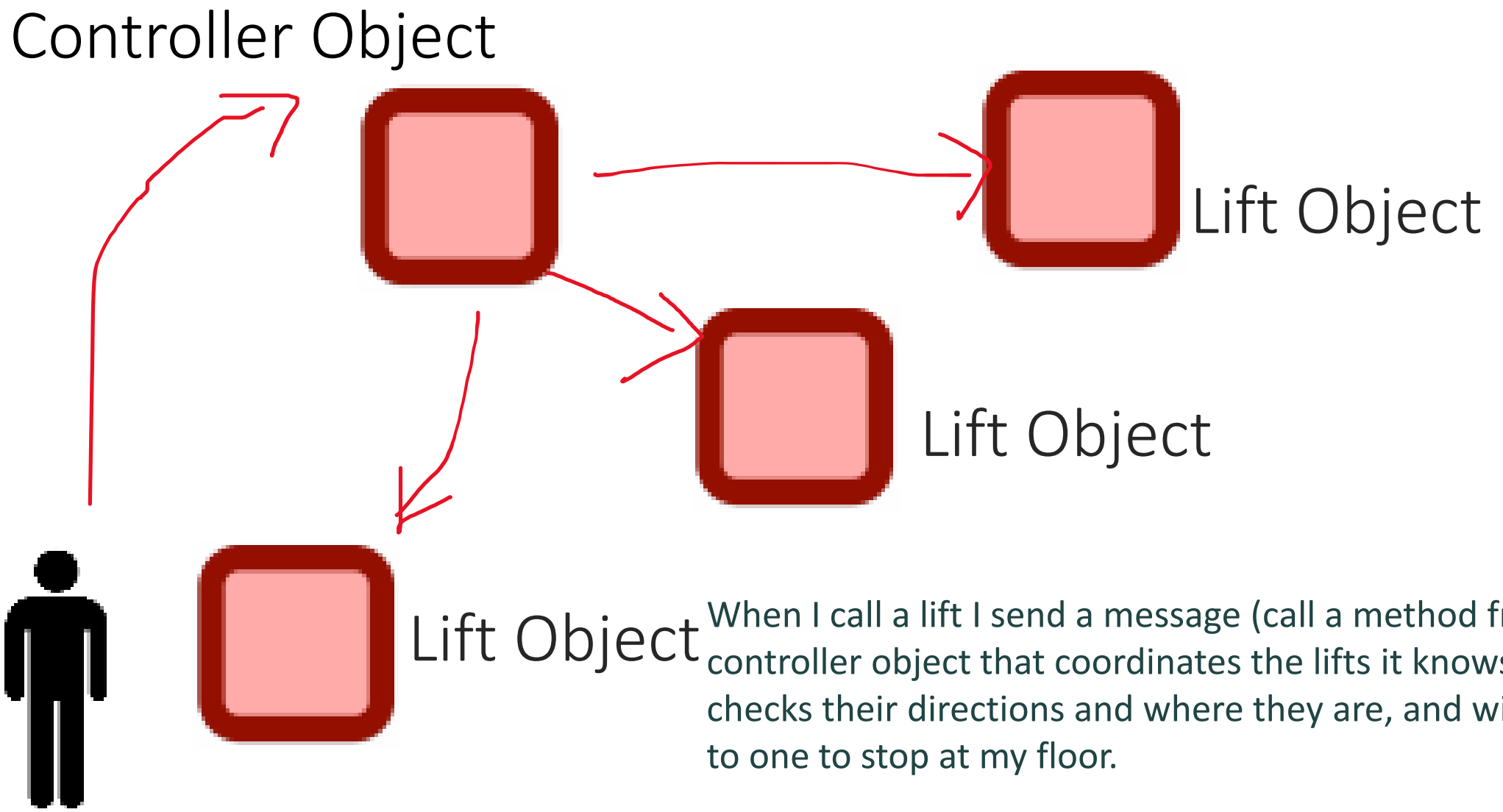
Controller Class

- 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.

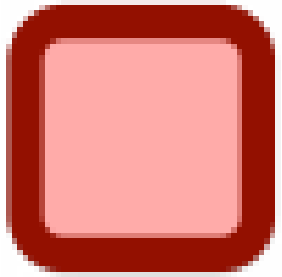
Lift Class

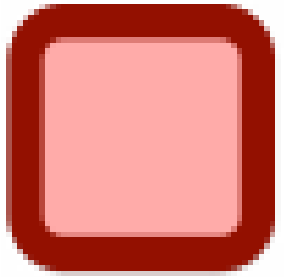
- Floors served
- Direction
- Moving/Stopped
- Stop request



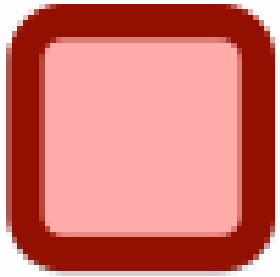
When I call a lift I send a message (call a method from) a controller object that coordinates the lifts it knows, checks their directions and where they are, and will ask to one to stop at my floor.

Controller Object

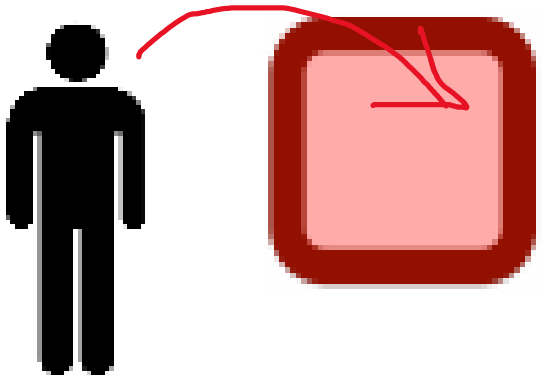




Lift Object



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

- **Maximize consistency and minimize coupling**

the hard part

- Trade-off between one object that does everything and objects that send too many messages