# S̃tudy Notes

Last updated on : March 10, 2014

# 1 Group

A group G is a finite or infinite set of elements together with a binary operation (called the group operation) that together satisfy the four fundamental properties of #1 closure, #2 associativity, #3 the identity property, and #4 the inverse property.

**1. Closure:** If A and B are two elements in G, then the product AB is also in G.

**2. Associativity:** The defined multiplication is associative, i.e., for all A,B,C in G, (AB)C=A(BC).

**3. Identity:** There is an identity element I (a.k.a. 1, E, or e) such that IA=AI=A for every element A in G.

**4. Inverse:** There must be an inverse (a.k.a. reciprocal) of each element. Therefore, for each element A of G, the set contains an element B=A$\hat{(}$-1) such that AA$\hat{(}$-1)=A$\hat{(}$-1)A=I.

**Group Theory:** Study of groups.

**Finite Group:** The group has a finite number of elements.

**Subgroup:** A subset of group, which is *closed* under *group operation* is called as subgroup.

**Cyclic Group:** Cyclic group is a group, that is generated by a single element. That is,

- It consists of a set of elemnets, with single invertible assosiative operation and
- a element g, such that, every other element in the group can be obtained by applying the group operation or its inverse to g.

Cyclic group is closed under addition, associative and has unique inverse.

**Homomorphism:** A map between two groups, which preserve identity and group operation is called Homomorphism.

**Isomorphism:** If a homomorphism has an inverse which is also an homomorphism, this is called as *isomorphism* and the two group is said to be isomorphic.

# 2 Android Services

We will see,

1. What is a Service?
2. Service Lifecycle
3. Permissions
4. Process Lifecycle
5. Local Service Sample
6. Remote Messenger Service Sample

## 2.1 Introduction

A Service is an application component, representing either

- an application's desire to perform a longer-running operation while not interacting with the user. *This corresponds to calls to* ***Context.startService()***
- to supply functionality for other applications to use. *This corresponds to calls to* ***Context.bindService()***

Service is

- Not a seperate process
- Not a seperate thread

Each service class must have a corresponding <service> declaration in its package's ***AndroidManifest.xml.***
Services can be started with ***Context.startService()*** and ***Context.bindService().***

## 2.2 Service Lifecycle

If someone calls *Context.startService()* ⟶ *onCreate()* ⟶ *onStartCommand(Intent, int, int)* ⟶ run until *Context.stopService()* or *stopSelf()* is called

*START_STICKY* is used for services that are explicitly started and stopped as needed.
*START_NOT_STICKY* or *START_REDELIVER_INTENT* are used for services that should only remain running while processing any commands sent to them.
*onDestroy()* method is called and the service is effectively terminated.

## 2.3 Permissions

Global access to a service can be enforced when it is declared in its manifest's <service> tag. By doing so, other applications will need to declare a corresponding <uses-permission> element in their own manifest to be able to start, stop, or bind to the service.
This can be bypassed by using *Intent.FLAG_GRANT_READ_URI_PERMISSION* and/or *Intent.FLAG_GRANT_WRITE_URI_PERMISSION* on the Intent, when using *Context.startService(Intent).*

# 3 Two step verification :

## 3.1 Factors of Authentication

| Factor | Explanation | Example |
|---|---|---|
| Ownership factor | Something that user has | ID Card, A hardware, Token |
| Knowledge factor | Something that user knows | Password, PIN |
| Inherence factor | Something that is inhere to user | Biometrics like fingerprint, retina etc |

## 3.2 Two step verification :

For a secured and positive authentication, atleast there should be two or all three factors involve in authentication.

**Single Sign On** A method to authenticate in a system which has multiple secured, independent systems. By this Single Sig On, a user logins in and autheticates one system and gains acess to all the systems.

# 4 Newline representation :

**Carriage Return [CR] :** Reset typewriters horizontal position to the far left.

**Line Feed [LF] :** Advance the paper by one line.

*ASCII* Based systems

A type writer typically needs both. Windows uses CR+LF. Linux uses only LF. MAC OS, prior to OS-X used CR alone.

IBM Pc's based on EBCDIC uses a special character called as *Newline* ***[NL]***

# 5 Turing Machine

A Turing machine can be thought of as a primitive, abstract computer. Alan Turing, who was a British mathematician and cryptographer, invented the Turing machine as a tool for studying the computability of mathematical functions. Turing's hypothesis (also known has Church's thesis) is a widely held belief that a function is computable if and only if it can be computed by a Turing machine. This implies that Turing machines can solve any problem that a modern computer program can solve. There are problems that can not be solved by a Turing machine (e.g., the halting problem); thus, these problems can not be solved by a modern computer program.

## 5.1 Components of a Turing Machine

A Turing machine has

- an infinite tape that consists of adjacent cells (or squares).
- On each cell is written a symbol. The symbols that are allowed on the tape are finite in number and include the blank symbol.
- Each Turing machine has it's own alphabet (i.e., finite set of symbols), which determines the symbols that are allowed on the tape.
- a tape head that is positioned over a cell on the tape. This tape head is able to read a symbol from a cell and write a new symbol onto a cell.
- The tape head can also move to an adjacent cell (either to the left or to the right).
- A Turing machine has a finite number of states, and, at any point in time, a Turing machine is in one of these states.
- A Turing machine begins its operation in the start state.
- A Turing machine halts when it moves into one of the halt states.

## 5.2 Opeartion of a Turing Machine

1. The Turing machine reads the tape symbol that is under the Turing machine's tape head. This symbol is referred to as the current symbol.

2. The Turing machine uses its transition function to map the following:

   {the current state and current symbol} = {the next state, the next symbol and the movement for the tape head.}

3. The Turing machine changes its state to the next state, which was returned by the transition function.

4. The Turing machine overwrites the current symbol on the tape with the next symbol, which was returned by the transition function.

5. The Turing machine moves its tape head one symbol to the left or to the right, or does not move the tape head, depending on the value of the 'movement' that is returned by the transition function.

6. If the Turing machine's state is a halt state, then the Turing machine halts. Otherwise, repeat sub-step #1.

# 6 NP Complete

Decision problem : Any problem with an answer, YES or NO.

**P :** A decision problem that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

***Example :*** Given a graph connected G, can its vertices be colored using two colors so that no edge is monochromatic. Algorithm: start with an arbitrary vertex, color it red and all of its neighbors blue and continue. Stop when you run our of vertices or you are forced to make an edge have both of its endpoints be the same color.

**NP :** A decision problem where instances of the problem for which the answer is yes have proofs that can be verified in polynomial time. This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

***Example:*** Integer factorization is NP. This is the problem that given integers n and m, is there an integer f with 1 ¡ f ¡ m such that f divides n (f is a small factor of n)? This is a decision problem because the answers are yes or no. If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with 1 ¡ f ¡ m and claim that f is a factor of n(the certificate) we can check the answer in polynomial time by performing the division n / f.

**NP-complete :** An NP problem X for which it is possible to reduce any other NP problem Y to X in polynomial time. Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X if there is a polynomial time algorithm f to transform instances x of X to instances y = f(x) of Y in polynomial time with the property that the answer to x is yes if and only if the answer to f(x) is yes.

***Example:*** 3-SAT. This is the problem wherein we are given a conjunction of 3-clause disjunctions (i.e., statements of the form

(x_v11 or x_v21 or x_v31) and (x_v12 or x_v22 or x_v32) and ... and (x_v1n or x_v2n or x_v3n) where each x_vij is a boolean variable or the negation of a variable from a finite predefined list (x_1, x_2, ... x_n). It can be shown that every NP problem can be reduced to 3-SAT. The proof of this is technical and requires use of the technical definition of NP (based on non-deterministic Turing machines and the like). This is known as Cook's theorem.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).

**NP-hard :** Intuitively these are the problems that are even harder than the NP-complete problems. Note that NP-hard problems do not have to be in NP (they do not have to be decision problems). The precise definition here is that a problem X is NP-hard if there is an NP-complete problem Y such that Y is reducible to X in polynomial time. But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

The halting problem is the classic NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one.

**P = NP :** This the most famous problem in computer science, and one of the most important outstanding questions in the mathematical sciences. In fact, the Clay Institute is offering one million dollars for a solution to the problem (Stephen Cook's writeup on the Clay website is quite good). It's clear that P is a subset of NP. The open question is whether or not NP problems have deterministic polynomial time solutions. It is largely believed that they do not. Here is an outstanding recent article on the latest (and the importance) of the P = NP problem: The Status of the P versus NP problem.

The best book on the subject is Computers and Intractability by Garey and Johnson. My favorite NP-complete problem is the Minesweeper problem.