Mathematical Induction

- *Deductive reasonin* ties the whole of mathematics. For example, take this problem from highschool: solve for y where $y = x^2 - 8$ and $x = 10$. We are using the information given by the equation and x to *deduce* the value y.
- Deductive reasoning in mathematics is given in the form of *proofs*.
- Unproven hypothesis that is known to hold = *conjecture*.

An example of deductive reasoning:

1. It will either rain or swow tomorrow. It is too warm for a snow. Therefore, it will rain tomorrow.

The argument universe is important.

Think of argument as an environment where an arguer assumes the truth value of premises and argues for a given conclusion. That means the "truth" of the statements in the universe is not considered and what matters most the *validity* for the logic flow.

We talk of premise in the context of arguments. Otherwise it is simple statements and compound statements. The truthiness of of compound of statement depends on the truthiness of its component statements and the logical connectives between them.

Example analyze the logical form of the following statement:
Either Bill is at work and Jane isn't, or Jane is at work and Bill isn't.

$$(P \wedge \neg Q) \vee (Q \wedge \neg P)$$

General form of dedcutive reasoning:

1. Logically connected statement or premise. It is more interesting when the connective is OR or IMPLIES.
2. A premise assumed to be true or false.
3. A conclusion.

Proof. Here is my proof:

$$a^2 + b^2 = c^2$$

□

Truth tables

We talk of premise in the context of arguments. Otherwise it is simple statements and compound statements. The truthiness of of compound of statement depends on the truthiness of its component statements and the logical connectives between them. [1]

[1] This is very important—context

P	Q	$\neg P$	$\neg Q$	$P \vee Q$	$P \wedge Q$
T	F	F	T	T	F
T	T	F	F	T	T
F	T	T	F	T	F
F	F	T	T	F	F

OR can be both inclusive (P or Q, or both) or exclusive (P or Q, not both). keep in mind. In mathematics, we all ways mean inclusive OR.

Writing (English)

Grammar

The most important part of a sentence that I need to keep an eye out for is the verb phrase[2]. Verb phrases are the heartbeat of a sentence. Noun phrases draw power from verb phrases. Noun phrases derive meaning from verb phrases. Almost everything else in a sentence is a modifier. One word may belong to many parts of speech. There is no function like relationship between words and parts of speech[3]. English follows the SVO (subject–verb–object) sentence pattern. The placement of words in a sentence is a more reliable way to tell the function of that word in a sentence.[4] For example, take a look at the following two sentences:

- *Winter* is harsh.
- Thomasia *winters* in Tuscon.

The same word *winter* represents two different parts of speech in each of the above sentences. In the first sentence[5], it is used as noun phrase that functions as a subject. In the second sentence, it is used as a verb. Verbs have the unique ability of "travelling" through time (present, past, and future) which we call *tense*. Verb is the only part of speech that can encode/decline tense in its form.

- Todd *arrives* in Tuscon.
- She *had* a drink.
- She *has* the baby.

In the above set of sentences, the verb phrases carry both the action and the actions period. Verb phrases are a combination of auxiliaries and other verbs. The combination accomplishes showing time and mood that a single word is not able to. For example:

- Kaput *ate* before leaving the house.

[2]The noun phrase modifier *that I need to keep an eye out on* is relative clause modifier, and the main verb phrase is *is*

[3]*is* is acting like a link verb between the pronoun *There* and *no function like relationship…* which is a subject complement

[4]This sentence is a litte more tricky to analyze, but form wise, it is similar to the sentence *The palcement of cars in a garage is great*. Its form is thus $NP - LV - SC$, and *of words in a sentence* is a prepositional phrase

[5]*In the first sentence* is a whole sentence modifier.

- Kaput *should have eaten* before leaving the house.
- By the time we get there, he *will have been* ***blowing up*** at the referee for quite some time.

The single word verb phrase *ate* conveys the simple past and *to eat* in its conjugation. the multi-word verb phrase *should have eaten* conveys an additional mood *should have* with its root action. *blowing up* is *verbal phrase* that is made up of two words but actually represents a new word with a different meaning. single verb phrases can form sentence. *Stop!* is a single verb phrase sentence. some verbs are transitive:

Pat sent *Chris* the message.
Julia *smells* bad.
Lisa *is smelling* to Pizza.
Mr. Becker *is* a big man.

verbs can be transitive or non transitive. transitive verbs can take objects. in the above sentence *the message* is direct object while *Chris* is indirect object. *linking verbs* tell more about their subjects. *smells* in the second sentence is a linking verb telling us more about *Lisa*. what comes after the linking verb is called the *subject complemnt*. subject complements are of two kinds: they can be *modifiers* or they can be other noun phrases. a rule of thumb to identify nouns is to check if the noun can take an article. articles are a type of determiners. determiners: *the*, *that*, *some*, *some*, *a*, ...

$$NP = (D + M_0 + N + M_1) + C + NP$$

where NP is a noun phrase, D is a determiner, M_0 is a modifier before the noun, M_1 is a modifier after the noun, C is conjuction, and N is a noun. *Some big, white, dependable* ***chickens*** *besides the red wheelbarrow and a delicate glaze of* ***rainwater***.

The squeamish are nervous.
The enourmous, fancy **Taco** is thrown away.
The **man** *in yellow hat* kicked the lady
A disastrous **event** *that left many dead*...

traditionally, *squeamish* is used as an adjective but its placement in the above sentence makes it work as noun representing the subject. in the second sentence *The* represents the determiner, *enourmous, fance* is the modifier and *Taco* is the root subject.

modifiers modify verb and noun phrases. verb phrase modifiers may appear before the verb or after the verb. single word modifiers end wih *-ly* but not always. example of multi word modifier is *prepositional pharases*. example below sentence 4.

The gazelle runs *fast*.
Sammy sings *pretty*.
Sam sings *good*.
Sam sings *well*.
Elyssa fished *under the full moon*.

Frita fished *while her friend cooked.*
The incomparable June had *wisely* hunted *only on Thursdays*.
The suspects *often* look suspicious.

the noun phrase is the most modifyable part of a sentence. noun phrase modifiers can appear before or after the noun.

This *cold, super sweet, photogenic banana* **ice cream** *on the table (last night's table desert) forgotten after the party and sitting in the sun and which we really did mean to go back to* is *ruined*.

one of the modifiers that come after the noun, *on the table* is a prepositional phrase. *(last night's table desert)* is an appositive. *forgotten after the party* is a participal phrase made of verbs, *sitting in the sun* is a participal phrase, made of verbs. *which we really did mean to go back to* is a relative clause noun modifier, and *ruined* that comes after the linking verb *is* is a subject complement modifier.

of the noun clauses we listed above two of them deserve attention: *relative clauses* and *complement* clauses. we modify noun phrases with relative clauses by adding another cluase talking about the same subject. for example the following clauses (independent) all have the same subjects. relative clause are introduce by relativizers like *who*, *which*, *that*, *whom*, *whose*.

- *The owl* caught the squid.
- The mouse fears *the owl*.
- *The owl* hoots every night.

we can use one of the sentence above as a relative clause modifier to produce:

- The owl *that the mouse fears* caught the squid.
- The owl *that hoots every night* caught the squid.
- The mouse fears the owl *that caught the squid*.
- The mouse fears the owl *that hoots every night*.
- The owl *that caught the squid* hoots every night.
- The owl *that the mouse fears* hoots every night.

complement clauses are somewhat similar to relative clause modifiers. relative clauses added while complement clauses are complement clauses are necessary for full information. complement clauses are usually used with noun phrases like: *The idea*, *The thought*, *The recommendation*, …new. example:

Complement: The recommendation *(that) we go to the park*…
Relative: The kids *(that) went to the park*…

complement clauses can form noun phrases on their own. for example:

Mable shouted *(the message) that she was ready for the debate*.

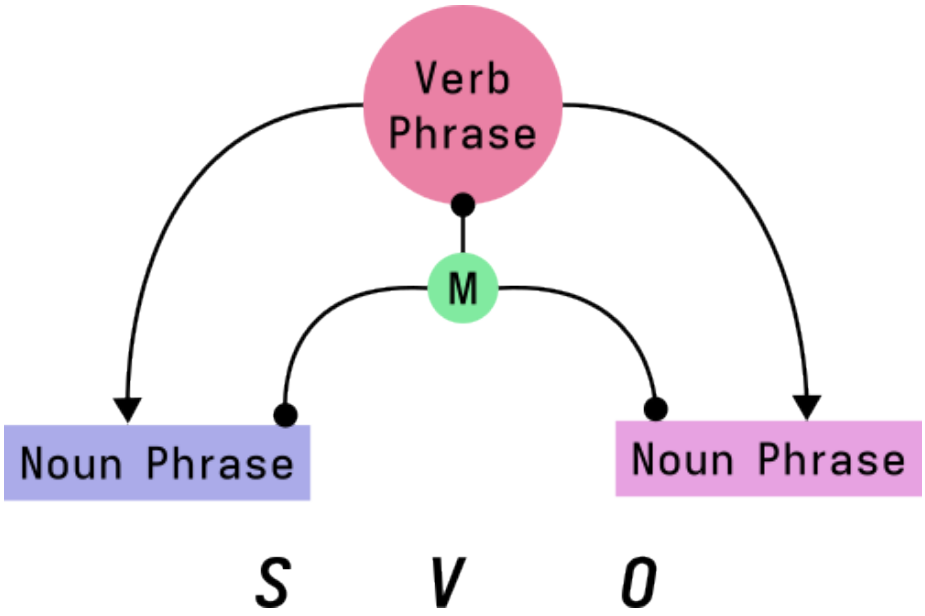

Figure 1: Typical sentence structure

(the message) that she was ready for the debate is used as a complete noun pharse acting as the object to *shouted*.

modifiers that sentences as a whole are called sentence modifiers. sentence level modifiers usually appear at the beginning of a sentence and are separated by a comma.

1. *Understandably*, Sam decided not to come.
2. *On Thursday night*, Raymond bereated Madeline.
3. *Motivated by fear*, the squirrel scampered.
4. *Please*, wear you masks.
5. *Astoundingly*, the ice cream brought about world peace.
6. *Under duress*, we confessed to dismantling the planes.
7. *Responding to the news*, the detective threw a fit.
8. *When I was a child*, my father gave me licorice.

sentence number 3 is an example of multi-word modifier. sentence number 6 is an example of sentence modifier that is prepositional phrase. sentence number 7 is an example of sentence modifier that is participal phrase. the place of modifiers matters a lot.

- *Really*, the old touscan shrieked loudly at the miserable goose.
- The *really* old touscan shrieked loudly at the miserable goose.

- The old touscan *really* shrieked loudly at the miserable goose.
- The old touscan shrieked *really* loudly at the miserable goose.
- The old touscan shrieked loudly at the *really* miserable goose.

sentence can be made out of single verbs *Stop!*, single independent clause that contains noun phrase and verbe phrase *Each of the students leaving the classroom are pathetic loosert indefinately.* or from subordinated and independent clauses *After I told him to cut it out, Schmitt began to make more scene.*. Subordinating clauses begin with words known as subordinatiing conjunctions. *After*, *Because*, *Although*, *Despite* are example of subordinating conjunctions.

{After the miserable party ended}, [Georgio boldly declared that he was the checkers champion], [but Lydia, incensed by the presumption, rejected his claim and proposed a board game coup[6]] {because she saw her opportunity to prevail[7]}. in the above complex sentence, subordinated clauses are in {} and independent clauses are in [].

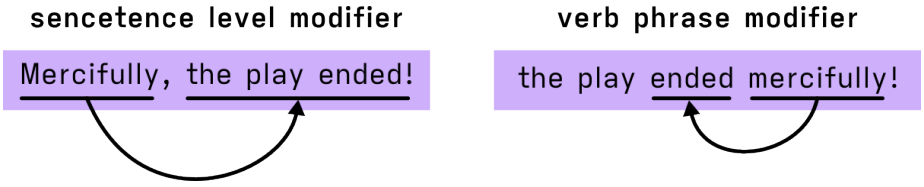

Figure 2: Sentence level and verb phrase modifiers

Comma usage recorded thus far:

1. After sentence level modifiers.
2. After subordinated clauses.
3. Between independent clauses conjugated with one of FANBOYS.

there is no agreed upon idea of what a paragraph should contain. paragraph is what start on indented line and ends a line white space. aggregation of words into whitespace separeted bunches makes paragraphs a contained unit of language. readers expect some from of structure from each of our paragraphs. *Focus* is the most expected from the paragraphs we write. each paragraph should talk about one idea only. *Topic sentences* at the start of a paragraph orient our readers on what is to follow in the paragraph so they know what expect and can follow easily. we can think of topic sentences as labels for a paragraph. human have a small working memory. make paragraph shorters and concise. a good *flows*. By flow, we mean sentences in the paragraph chain together to create effortless reading experience.

[6] *rejected his claim and proposed a board game* is a single verbal phrase with two objects attached.

[7] subordinated clauses can come after the independent clause

Infinitive		*wollen*
Present tense	ich	will
	du	willst
	es	will
	wir	wollen
	ihr	wollt
	Sie	wollen
	sie	wollen
Past tense	ich	wollte
	du	wolltest
	es	wollte
	wir	wollten
	ihr	wolltet
	Sie	wollten
	sie	wollten
Participle		gewollt

Table 1: *wollen* conjugations.

Deutsch

Phonology

Phonolgy is grammar of the sounds of a language, but phonetics is the study of human produced sound for its own sake.

the text bellow contains the translation of Rammsteins hit ich will.

```
1  ich will
2  ich will
3  ich will
4  ich will
5  ich will
6  ich will
7  ich will
```

ich is the first singular personal pronoun in the nominative case. the nominative case marks the subject of a verb. the ohter cases of the first person singular personal noun are *mich*, *meiner*, *mir* for accusative, genitive, and dative cases repectively. *will* is the singular first person present conjugation of *wollen*. *ich will* translated to english means *i want*.

```
1  Ich will dass ihr mir vertraut
2  (Ich will) Ich will dass ihr mir glaubt
3  (Ich will) Ich will eure Blicke spüren
4  (Ich will) jeden Herzschlag kontrollieren
5  (Ich will) eure Stimmen hören
```

	Masculine	Feminine	Neuter
Nominative	jeder	jede	jedes
Accusative	**jeden**	jede	jedes
Genitive	*jedes*	jeder	jedes
Dative	jedem	jeder	jedem

Table 2: Declension of *jeder*

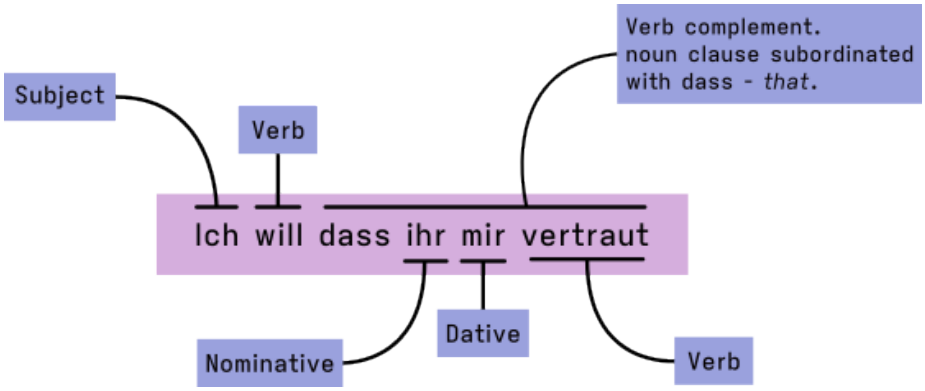

Figure 3: Verb structure for the first verse

6 (Ich will) Ich will die Ruhe stören
7 (Ich will) Ich will dass ihr mich gut seht
8 (Ich will) Ich will dass ihr mich versteht

dass is a subordinating clause introducer similar to english's *that*. *eure* is the genitive of the second person plural *ihr* (you (all)). *jeden*[8] is a determiner that corresponds to english *each*, *every*. *kontrollieren*

1 I want that you (all) trust me
2 (Ich will) I want that you (all) believe me
3 (Ich will) I want your glancing look.
4 (Ich will) Each heartbeat controlling.

Grammar

All german nouns are "gendered". the genders are: neutral, male and female. but the actual gender of the noun has nothing to do with its gender especially for inanimate objects. german nouns are to be memorized with the article reflecting the gender. These are *der*-nouns (masculine), *die*-nouns (feminine) and *das*-nouns (neuter). Examples: *das Cafe*, *der Flughafen*, *der Banhof*, *das Restaurant*, *das Hotel*, *die Botschaft*, *die Bank*, *die Zigarren*, *der Wein*, *das Bier*, *der Kaffe*, *der Tee*, *die Milch*, *das Wasser*, ...It makes no sense for wine to a masculine gender.

[8] except that jeden (rather than jedes) is usual in the genitive singular masculine and neuter if the following noun has the ending -(e)s, e.g. *am Ende jed* ***en*** (less frequent: *jed* ***es***) *Abschnitts.*

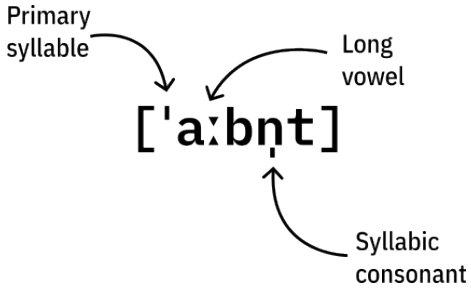

Figure 4: IPA symbols.

Personal pronouns

Nominative	Accusative	Genitive	Dative
ich	mich	meiner	mir
du	dich	deiner	dir
er	ihn	seiner	ihm
sie	sie	ihrer	ihr
es	es	seiner	ihm
wir	uns	unser	uns
ihr	euch	euer	euch
Sie	Sie	Ihrer	Ihnen
sie	sie	ihrer	ihenen

Table 3: Personal pronouns in Deutsch.

Unlike Enlgish verbs which are only conjugated for number, and tense, German verb conjugation depends on number, gender, person, mood, and tense. German verbs can be regular or irregular in their conjugation. Finite verbs must agree with the subject, unlike non-finite verbs.

The “principal parts” of a verb are:

1. Infinitive form.
2. Past tense form.
3. Past participle form.

Based on conjugation I need to worry about:

1. Weak verbs.
2. Strong verbs.
3. Irregular verbs.
   (a) Irregular weak verbs.
   (b) Irregular strong verbs.
   (c) The modal auxiliary verbs and *wissen*.
   (d) The verbs *haben*, *sein*, and *werden*.

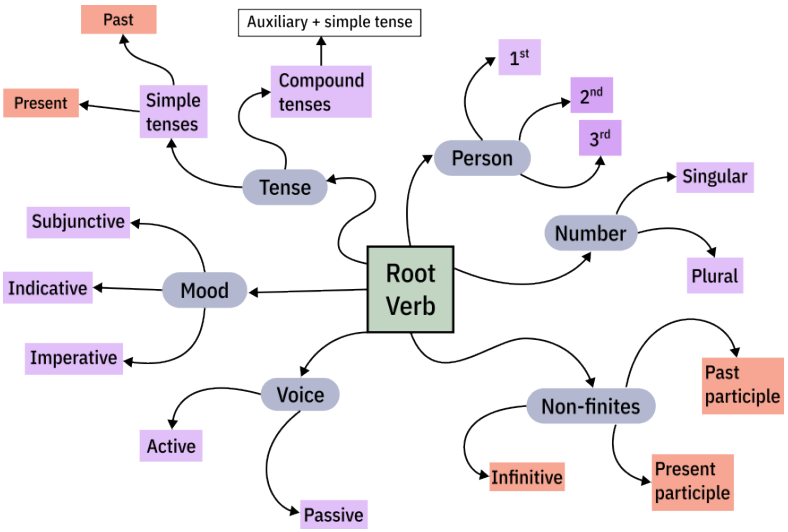

Figure 5: German verb conjugations

Java notes

Type theory interlude

Subtyping

In programming language theory a *subtype* referes to a a type that is related to another type, also called the *supertype*, by some notion of *substiutability*, meaning that program elements typically functions (or subroutine) written to operate on the supertype can also operate on the subtype. If S is the subtype of T, the subtyping relation is often written as $S <: T$, to mean that any term of type S can safely be used in places that expect type T to be present. For example in Java, every type except for primitive types is the subtype of `Object` class. Or stated mathematically:

$$E <: O$$

Where E is every other class and O is the `Object` class.

Covariance, contravariance, and invariance

Variance refers to how the subtyping of more complex types is related to subtyping between the component types. Complex types include: generics, functions, and collections types like arrays, maps, and linked lists. For example, should `List<Cat>` be the subtype of `List<Animal>` give that type `Cat` is the subtype of type `Animal`? Does

$$L[C] <: L[A]$$

hold given that

$$C <: A$$

for a given programming language like Java? Since Java supports generics, which allow the programmer to extend the type system with new type constructors (parametric polymorphism), which raises the question should `ArrayList<File>` be the subtype of `ArrayList<Object>` (*covariant*)? Java uses *use-site* annotation to describe the varaince of the generic type constructors. *declaration-site* annotation is used by C#, Kotlin, and Scala.

Within a type systm of a programming language, a type rule or a type constructor is:

- *covariant* if it preserves the ordering of types ($\leq$), which orders types from more specific to more generic: If $C <: A$, then $I[C] <: I[A]$.
- *contravariant* if it reverses this ordering if $C <: A$, then $I[A] <: I[C]$.
- *bivariant* if both of this apply. (i.e., if $C <: B$, them $I[C] \equiv I[A]$).
- *variant* if covariant, contravariant, or bivariant.
- *invariant* or *nonvariant* if not variant.

Package management

Classes live inside packages. a package declared using `package ...;` classes living in the same package can see each other. package *classpath* and directory structure of the source must match each other. always think of two contexts the classpath context and source context. why should java source directory tree and package name classpath match? answer here. relationship with maven group id and artifact ids?

The language

Non wildcard (`G<?>`) parameterized types are invariant in Java, i.e, there is no subtyping relationship between `List<Cat>` and `List<Animal>`. [9]

Java does not suffer from template bloat like C++ does. Why? That is because C++ creates a new type for every template instantiation. For example:

```
template <typename T>
void print(T arg) {
    // ... implementation
}

print<int>(30);
print<const char*>("Hello");
```

Essentially generates two copies of the function print with the type paramenters resolved: `printInt` and `printConstPtrToChar` which generates bloat during compilation. Java unlike C++ generates just one type for each generic type with the generic type thrown away, which call *type erasure*.

```
public <T> void print(T arg) {
    // ... implementation
}
```

[9] more on this.

becomes just one function with type parameters replaced with `Object` type.

```
    public void print(Object arg) {
        // ...implementation
    }
```

Dependency injection

Dependency injection is a design pattern applied to classes with members so that the member are initialized outside the class itself. For example, the following code:

```
package io.github.termitepreston;

public class RealBillingService implements BillingService {

    @Override
    public Receipt chargeOrder(PizzaOrder order, CreditCard
        creditCard) {
        CreditCardProcessor processor = new
            PaypalCreditCardProcessor();
        TransactionLog transactionLog = new DatabaseTransactionLog()
            ;

        try {
            ChargeResult result = processor.charge(creditCard, order
                .getAmount());
            transactionLog.logChargeResult(result);

            return result.wasSuccessful()
                    ? Receipt.forSuccessfulCharge(order.getAmount())
                    : Receipt.forDeclinedMessage(result.
                        getDeclinedMessage());
        } catch (UnreachableException e) {
            transactionLog.logConnectException(e);
            return Receipt.forSystemFailure(e.getMessage());
        }
    }

}
```

`RealBillingService` class depends on two internally constructed objects inside the `chargeOrder` function. If we wanted to test `chargeOrder`, we will have to charge from a real Paypal account which is impractical. To solve this we could use:

- Using a Factory class but we would have to reset this global factory after each test.
- Passing every dependency to constructor manually. using method we can remove `setUp` and `tearDown` methods from our test code. further more expose the dependency in the api signature.
- Using a dependency injection framework.

Java is getting better and better with each release. Keep the the features listed below in mind when working with a new java project.

- better `switch` blocks.
- a smarter `instanceof` operator.
- Records with autogenerated getters, setters, and to string.
- Text blocks.
- sealed classes.

Java security primitives

Java uses serveral classes and interfaces from core java packages to thrid party libraries to help with the control of *access to information*. *Principal* interface represents an abstract notion of a principal, which can be used to represent any entity, such as an individual, a corporation or a login id. Essentially, anything with a name (that name could be a user id from user database) is principal. A *Credential* is a piece of document that details the qualification, competence, or authority issued to an individual by a third party with a relevan defacto authority assumed competence to do so. Examples of credentials include academic degrees, passwords, security clearance, badges, passwords, user names, keys, and certifications. *Subject* `class` represents a *grouping* of related information for a single entity, such as a person. Such information includes subjects indentities as well as security related attributes (passwords, cryptographic keys, for example.) Subjects may potentially have multiple indentities. Each identity is represented as a `Principal` within the `Subject`. For example a `Subject`, that happens to be a person, Alice, might have two principals: on which binds "Alice Bar", the name of her driver license, to the `Subject`, and another which binds "999-99-999", the number of her student identification card, to the `Subject`. Both `Principal`s refer to the same `Subject` even though each has different name.

```
package java.security;

public interface Principal {
    // ...
    String getName();
    boolean implies(Subject subject);
    // ...

}
```

Important Java foundations

The Eclipse foundation and Apache foundation contribute a great deal to the advancement of the Java ecosystem. Besides that Red Hat and Oracle are commercial companies engaged in the development and support of Java Platform.

Jakarta EE

Jakarata EE also previously known as Java Enterprise Edition is a set of *Specifications* that extend Java SE with specifications for enterprise features such as distributed computing and web services. Jakarata EE defined

by its specification, and its specification defines APIs and their interaction. Jakarta EE was maintained by Oracle corporation who later transffered its development to Eclipse foundation was renamed from Java EE to Jakarta EE because Oracle owns the trademark for *Java*.

OSGi

OSGi specification describes a modular system and service platform for Java that implements a complete and dynamic component model, something that does not exist in standalone Java/VM platforms. In enterprise settings typical Java application is not packaged as jar and launched from its main function using the system installed java executable, rather than that the enterprise system provides a java platform that *always* runs in which application bundles are loaded and unloaded with out restarting the application server. OSGi architecture has the following components:

1. *Bundles* are normal JAR components with extra manifest headers.
2. *Services* layer connects bundles in a dynamic way by offering a publish-find-bind model for POJIs and POJOs.
3. *Service registry* the application programming interface for management services.
4. *Life-cycle* the application programming interface for lifecycle management (insatll, start, stop, update, uninstall) for bundles.
5. *Modules* layer defines encapsulation and declaration of dependencies (how bundles can import and export code).
6. *Security* layer that handles the security aspects by limiting bundle functionality to pre-defined capabilities.

Apache Felix is implementation of the OSGi specification.

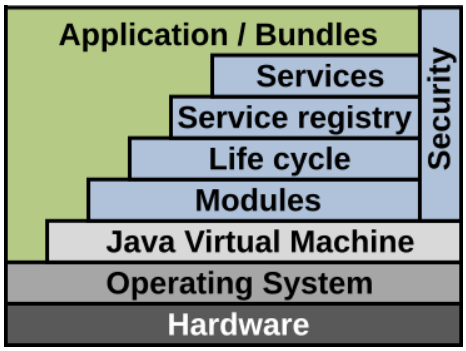

Figure 6: OSGi architecture.

Glassfish

JAX-RS

Jakarta RESTful Web Services also called *JAX-RS* is a set of interfaces and annotations included in Java EE platform that help in writing REST applications. Since JAX-RS is just a collection of interfaces and annotations it just defines an API. RestEasy from Red Hat, Jersey from Eclipse foundation and Apache CXF are the libraries implementing the API.

Root resource classes are POJOs that are annotated with `@Path` have atleast one method annoted with `@Path` or a resource method designator annotation such as `@GET`, `@POST`, `@PUT`, or `@DELETE`. Resource method are methods of resource class that are annotated with resource method designator.

Java world is full of specifications and implementations (some of which are reference). Language features such as interfaces, annotations and abstract base classes aid writing specifications in java code.

The server

Traccar server is a 3 layer monolithic application where each layer handles a specific functionality[10]. The top layer in the application receives incoming request from mobile and application clients as well as both http and websocket connection. The middle layer of the application processes these requests and performs business logic before the third layer persists or reads objects to a database.

The top layer of Traccar is composed of a set of servlets. One of the servlets handles incoming REST API (Application Programming Interface) calls from mobile and web clients. The other servlets handle websocket communication with tracking devices and serving media files for web clients. Since almost all of our modification is related to business logic and authorization related stuff, we will focus on the REST API servlet.

The REST API servlet implements the Jakarta RESTful Web Services (JAX-RS) specification. JAX-RS specification provides its users with annotations to decorate their classes. Traccar server uses Eclipse Jersey as the implentation library of JAX-RS. REST API resources are bound to handler classes with annotations. Each handler class is annotated with REST API resource *path*. Annotated functions in the resource class handle specific HTTP methods. This layer of the application is commonly called the controller. Jersey takes FasterXML's Jackson JSON serialization/deserializtion library in its filter chain to convert incoming JSON payloads to the application objects.

Before each request is received by the resource handler classes, it passes through a class that implents JAX-RS' `ContainerRequestFilter` interface. This interface contains the `filter` function. The `filter` function allows us to perform custom logic on all requests before they are matched with and dispatched to our resource handler classes. Our implementation of the interface checks if each request contains a JSON Web Token (JWT) payload in

[10]If this is the topic sentence, then the rest of the paragraph should be about some sort of layering.

the `Authorization` header, and If the header is present it tries to decode it and retrieve a valid Firebase user using Firebase Admin library functions.

Another function of our class implementing `ContainerRequestFilter` is to inject objects that are usefull for processing incoming requests. It injects a `SecurityContext` object for use by our resource handler classes. The injected object contains authentication information necessary for the logic inside our resource handlers. One such information is the user's unique identifier ID decoded from the JWT payload.

`Storage` class is also injected into the resource handlers. `Storage` interfaces with the underlying database to write and read application objects in memory. Connection to the underlying database is managed by HikariCP connection pool management library. We manage our connections with HikariCP because we want to be resource efficient. We become resource efficient by not recreating database connection for every request but by storing and reusing connections from older completed requests. By doing connection pooling, we reduce object creation/destruction overhead.

Traccar server uses Liquibase database migration tool to automatially initialize tables on startup. Liquibase reads table creation data from *schema* files written in XML. Each schema file is written in the from of *changesets*. changesets help us in the process of restoration of older database state. Traccar server uses a custom Object Relational Mapping (ORM) system. Each Traccar model class we want to persist is annotated with `@StorageName`. `@StorageName` helps the ORM to generate column names and table names using Java's support for runtime reflection. By using annotations and reflection, we can load database table rows to application memory objects of a classes that are annotated with `@StorageName` or insert objects that are annotated with `@StorageName("table_name")` to database rows. Each of the API handlers use instance of `Storage` class to read and write persistance data.

Data managers sit between `Storage` class and our resource handlers to facilitate the implementation of a more complex logic. This more complex logic is usually a code that shares functoinaliy with the websocket code.

Objects in Traccar server are linked with each other post creation. Special tables in the database store this linkage information. `Permission` model class helps when linking objects together.[11]. `PermissionService` class is also injected for use by resource handler classes. `PermisssionService` class handles various authorization related functionalities. Authorization related function include: check if user is admin, check if users can access reports, check if a give user can edit or update database objects, check if a given user owns certain model classes, etc…

SQL notes

Why did relational databases win out? Database is a *set* of related information[12]. Databases can be indexed with some *key*. A phonebook has letters on its margins. We can use the letters on the margins to find phonenumbers quicker. We call this process *indexing*. relational mode of data storage is the most commonly used. heirarchial mode of data storage was more popular in

[11] how?
[12] Learning SQL

the past. in heirarchial data storage scheme data is stored in a tree fasion. as an example of heirarchial data storage we can consider the case of bank account storage.

network based database store records with pointers to other records. user can extract data by traversing the linking pointers. relational database is the most common now. primary keys uniquely identify a row in relational table. primary keys need not be generated by the database management software. several columns can generate a unique key for each row. `fname` and `lname` in the database attached can be used as a primary key. primary keys made out of multiple columns are called *compound key*. *natural keys* are keys generated from the data itself. *surrogate keys* are keys appended by the database designer. database designers generate surrogate primary keys because no natural key can be used to identify a row. primary keys should never change once they are assigned. relational database tables include column whose value point to entries in other tables. column value that point to another table are called foreign keys. foreign keys perform the same function as pointers in network based database systems.the process of "lensing" into foreign key of table is called a *join operation*. storing redundant is not considered a problem in relational database system. we must be sure that redundant data stored has a reason to be redundant. cannonical data should be stored in one table. foreign keys (pointers) can be redundant. duplication of cannonical data causes for unreliable data. a single column should just include one data. this process of making cannonical data non redundant, storing single values in each column is called *normalization*. sql is a computer language that was specifically designed to manipulate relational databases. Edgar F. Todd described relational database systems. he proposed a language called DSL/Alpha for relational database systems. IBM created a language called SQUARE by refining edgar f. codds language. SQUARE was refined to SEQUEL. SEQUEL was shorted to SQL. The result of sql query is a table. a new table can be created in sql by simple storing the result set of an sql query. a query can use both permanent tables and result set of another query. sql is not an acronym for anything. sql is short for sequel. sql is a non procedural computer language. we do not tell the sql engine how to retreive our data. we describe the data we want to the sql engine and the sql engine retreives the data the most efficient way possible. sql statement types can be divided into three types. *schema statements* are used to define data structures stored in the database. *data statements* are used to manipulate data structures created by schema statements. transaction statements are used to begin, end, and rollback transactions. an example of a the schema statement `create table`:

- temporal data types. - constraints, especially foreign ones. - table joins.

```
create table corporation (corpid smallint, name varchar(30),
    constraint pk_corporation primary key (corp_id));
```

the above statement creates a table with two columns. the tables names is corporation. `corp_id` is identified as the primary key. primary keys are a type of constraint. `insert` is a data statement that is used to insert a row of data into a table. notice the single quote in insert.

```
insert into corporation (corpid, name) values (30, 'SpaceX');
```

another data statement is `select` statement. `select` statement is used to extract data from the database.

```
select (name, corpid) from corporation where corpid=30;
```

all database elements created with sql schema statements are stored in a set of special tables called *the data dictionary*. this set of metadata tables can be queried just like other sql tables.

several *clauses* make up a query statement. only `select` is mandatory for a query statement. `select` clause runs last after filtering, grouping and ordering operations. instead of just *selecting* rows from operation of other claues, for example `from` clause's, `select` clause can add synthetic rows to the result set. we can think of like the other clauses providing `select` for columns to select from. `as` introduces a column alias. `from` clause defines the tables used by a query, along with the means of linking tables together. `from` clauses are very important for table join operations.

```
-- select every column from language table for the result set.
select * from language;

-- select just language_id and name from language table for the
    result set.
select language_id, name from language;

-- select for 'synthetic' rows.
select
    language_id as id, -- id is a column alias for language_id
    'COMMON' as language_usage, -- 'synthetic' row
    language_id * 3.1415927 as lang_pi_value, -- 'synthetic' row
    upper(name) as language_name, -- 'synthetic' row
    version() as version -- or even this!
from
    language;
```

`from` clause can take various forms of tables:

1. *Permanent tables* created with `create table`.
2. *Derived tables* retured from a subquery and held in memory.
3. *Temporary tables* that are volatile tables held in memory.
4. *Virtual tables* created with `create view` statement.

```
select
    concat(cust.last_name, ', ', cust.first_name) as full_name
from -- from clause taking in a drived table.
    (select
        first_name,
        last_name,
        email
    from
        customer as c
    where
        c.first_name = 'JESSIE') as cust;
```

`from` clauses can also choose from 2 or more tables. by default from statement with out a join type will perfrom cross product for each row of all the columns of table A with all columns of table B. table joins can be chained together for more than two tables.

Clause name	Purpose
`select`	Determines which query to include in the query's result set
`from`	Identifies the tables from which to retrieve data and how the tables should be joined
`where`	Filters out unwanted data
`group by`	User to group rows together by common column values
`having`	Filters out unwanted groups
`order by`	Sorts the rows of the final result set by one or more columns

Table 4: Query clauses

```sql
select
    concat(cu.first_name, ' ', cu.last_name) as name, ct.city
from
    customer as cu
    inner join address as ad
        on cu.address_id = ad.address_id
    inner join city as ct
        on ad.city_id = ct.city_id;
```

one unique cutomer (by definition) can have multiple rental records. this, we call one-to-many relationship.
`group by` collects collects rows by equality of rows we provide. `count(column)` stores how many there are in a group.

```sql
select
    f.film_id, f.title, count(*) num_copies
from
    film as f
        inner join inventory as i
        on f.film_id = i.film_id
group by
    f.film_id, f.title;
```

`inner join` truncates rows that are not in both tables based on the condition we provided. to include rows of one table no matter what, we employ `outer join`.

```sql
select
    -- note: count(i.inventory_id)
    f.film_id, f.title, count(i.inventory_id) num_copies
from
    film as f
    left outer join inventory as i
    on f.film_id = i.film_id
group by f.film_id, f.title
order by f.film_id;
```

PostgreSQL

PostgreSQL is one of the most popular sql based database management systems. PostgreSQL is fully opensource. PostgreSQL is developed by Berkley University. installing posgresql on an Arch Linux machine creates a `postgres` user. `postgres` user is the only one allowed to interact with the database system. to change to user postgres use the following command. command work if `sudo` is installed. postgresql follows the client server model. postgresql server manages the datastore. postgresql server receives sql commands from clients and executes it against the datastore. servers and clients need not be on the same host. server client can communicate using TCP/IP.

```
sudo -iu postgres
```

then while we are user `postgres`, initialize the database cluster with locale and encoding we desire.

```
# initializes a database cluster.
# run as `postgres` unix user.
initdb  --locale=en_US.utf8 --encoding=UTF8 \
        -D /var/lib/postgres/data --data-checksums
```

PostgreSQL installs the following programs on our system: `psql` is a postgresql client that helps us interact with PostgreSQL from a terminal session. `createuser` is used to create PostgreSQL roles. `postgres` is used to manage the server process. `initdb` is used to initialize database clusters. `createdb` to create new database (not a cluster). `dropdb` to remove a database. commands from postgresql package must be run from the `postgres` unix user. postgresql has the concept of roles. roles in postgresql are a username password combo that is used to connect with a specific database. unix users and postgresql roles are a completely different concepts. a database client from a remote machine can use a database installed locally by providing the correct roles. management of the database server is perfromed by *unix* user `postgres`. postgresql comes with one default role `postgres`. `postgres` role is a super user role. super user roles can read and write to every database. super user roles can create and drop roles. linux user `postgres` and postgresql superuser role having the same name make things a little confusing. postgresql roles can be create from the terminal using `createuser` or from the interactive sql shell psql using sql statement: we can connect to to database using superuser role `postgres` from anywhere!

```
create role dbname;
```

postgresql db roles can be removed using:

```
drop role dbname;
```

Create a database for role `dbname`.

```
# create database named db for role dbname
# remember unix user is postgres
createdb db -O dbname
```

Then connect to the database using `psql`

```
psql db -U dbname
```

we will be presented with the following prompt.

```
psql (14.3)
Type "help" for help.

db=>
```

arrow instead of hash in `db=>` indicates we are a non superuser role.
contrast this with prompt for superuser role `postgres`:

```
psql (14.3)
Type "help" for help.

db=#
```

we can use `psql` command `\dt+` to list all available tables in a database.

Algorithms

What are algorithms?

An *algorithm* is is any well defined computational procedure that takes some value, as *input* and produces some value, or a set of values, as *output* in finite amount of time. Essentially algorithms are a set of procedures that transform input to outputs. [13]
Alternatively an algorithm can be described as a tool for solving a well defined *computational problems*. The statement of problem describes the desired input/output mapping for problem instances. The algorithm describes computational procedures for achieving the desired input/output relationship for all problem instances.
Example problem: sort a sequence of numbers in ascending order. Below is how we formally define *the sorting problem*.
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

- a correct alogrithm should produce the correct output for each input but also halt. (finite running time.)
    1. Prove that the algorithm actually *works*.
    2. Analyze the running cost of the algorithm especially interms of order of growth.
- The example of comparision between a slow implementation of merge sort with cost of $c_1 n \lg n$ and insertion sort with running cost of $c_2 n^2$ is illuminating.

C++ notes.

Smart pointers.

I should avoid using raw pointers whenever possible. Why?

[13] As defined in the Algorithms book.

- Declaration does not indicate whether they point to a single object or an array.
- Declaration does not tell us whether a pointer should destroy the object it is pointing at i.e. it is owning.
- There is almost no way to know whether to call `delete` or `delete []` from its declaration.
- Pass the pointer to a dedicated destroy function or just `delete` it? Hard to know.
- `std::unique_ptr<T>` encapsulates the single ownership concept.
- `unique_ptr` is the only creator and destroyer of an object.
- `std::shared_ptr<T>` described using people in a hall last one turns off the lights analogy. how?

```cpp
#include <memory>
#include <iostream>

using std::cout, std::endl;
using UniquePtrInt = std::unique_ptr<int>;

void takesUptr(UniquePtrInt uptr) {
    cout << "*uptr = " << *uptr << endl;
}

int main() {

    UniquePtrInt p { new int {30}};

    takesUptr(std::move(p));

    // p is nullptr
    // p has been "moved" from, so it is invalid.
    if(p)
        cout << "*p = " << *p << endl;

    return 0;
}
```

RAII

- Always prefer list initializations.
- Member declaration site initializations run before constructors.
- Constructor overloading is good.
- Constructors can `throw` exceptions and infact it is prefered to do so to "preserve" the class invariant.

Profiling and (micro)benchmarking

The real problem is that programmers have spent to much time worrying about efficiency in the wrong places and at the wrong times.[14]

1. Sampling profiling.
2. Instrumentation profiling.

And for benchmarking

1. Micro benchmarking
2. Macro benchmarking

imporatant `https://youtu.be/fHNmRkzxHWs?t=2122`

Algorithms

$$a = a + b$$

[14] Mathieu Ropert—youtube video

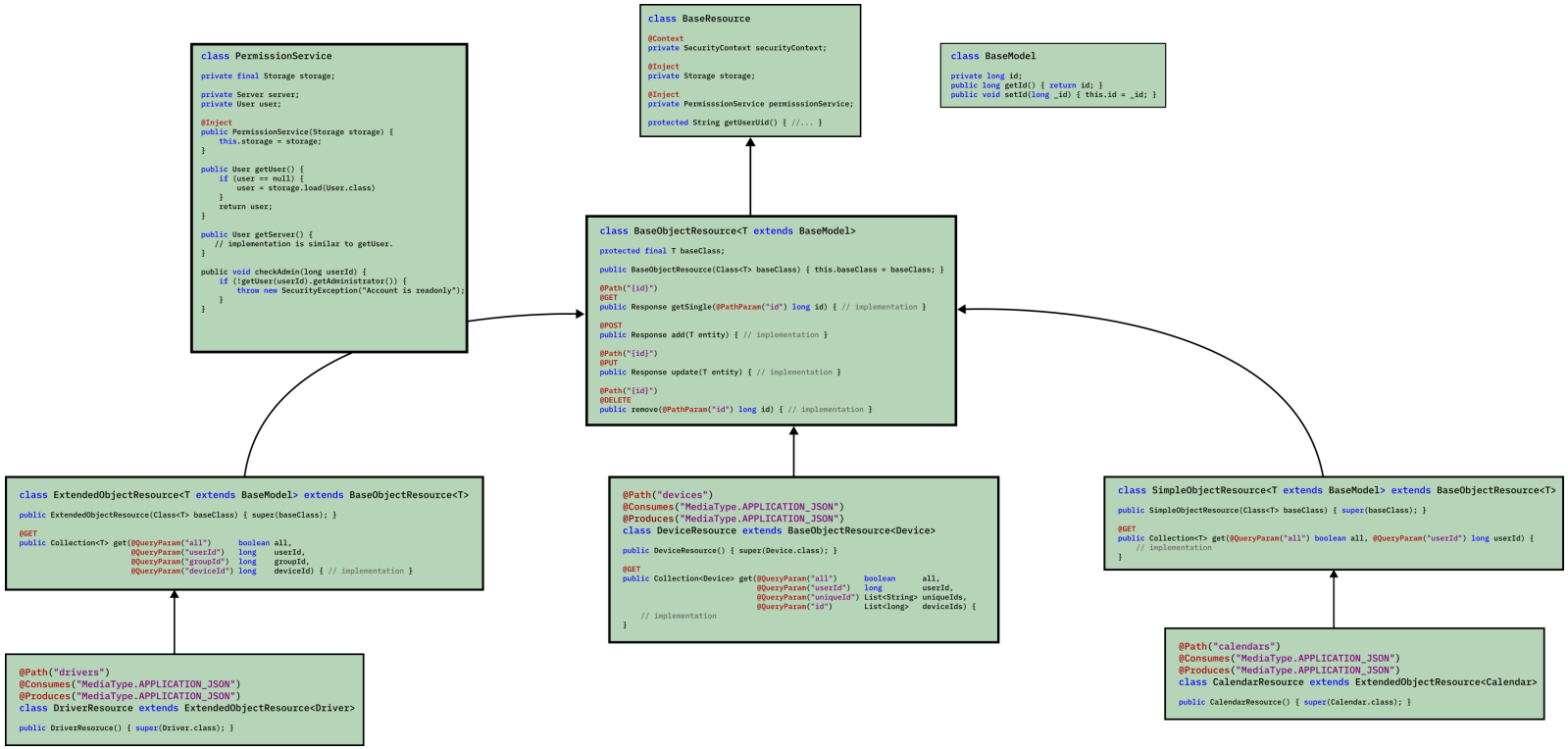

Figure 7: Traccar UML chart

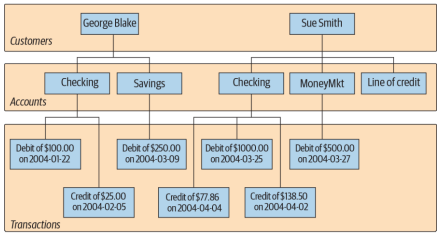

Figure 8: Heirarchial mode of data storage.

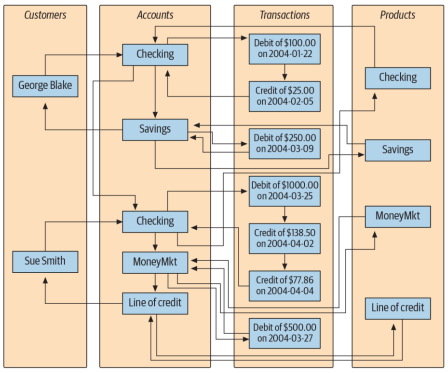

Figure 9: Heirarchial mode of data storage.

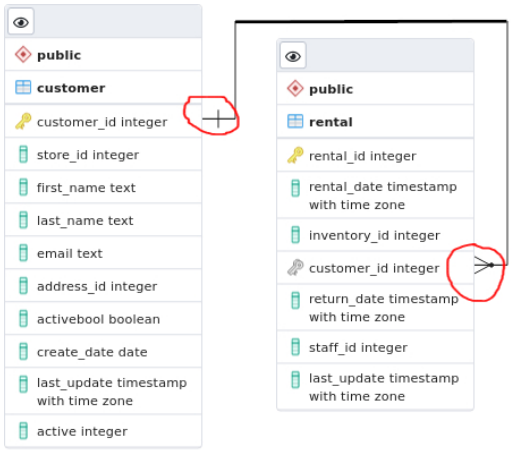

Figure 10: One to many relationship in tables