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.

$$(P \land \neg Q) \lor (Q \land \neg P)$$

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)

My most common problem while writing is my paragraphs become a sequence of sentences without much linking (coherence) between them. They call this the *the shopping list* paragraph.

- Reflexive repetition.
-

Deutsch

Phonology

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

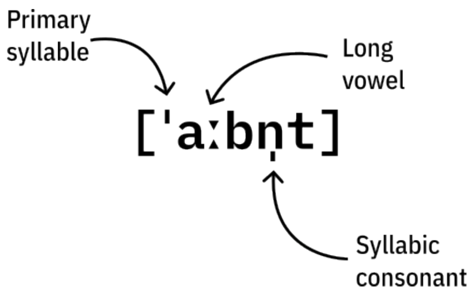

Figure 1: IPA symbols.

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.

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 1: 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*.

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

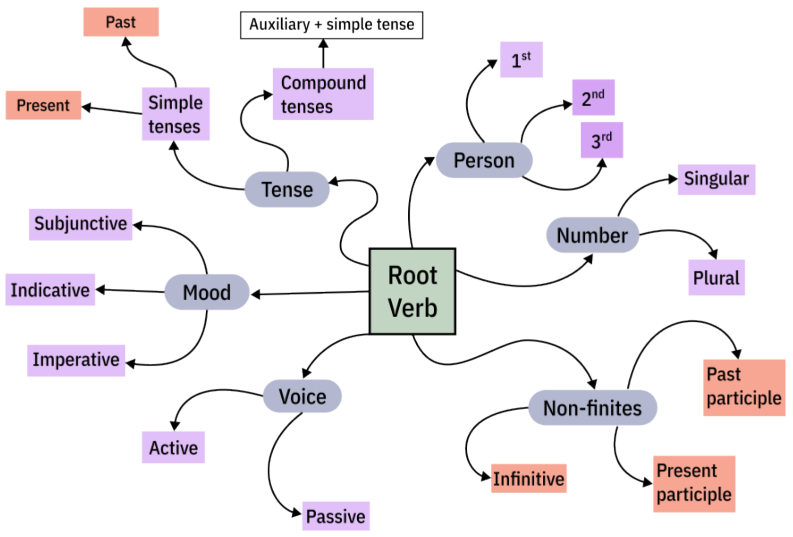

Figure 2: German verb conjugations

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 (≤), 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>`. [2]

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
}
```

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

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

[2]more on this.

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:

```java
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.

```java
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.

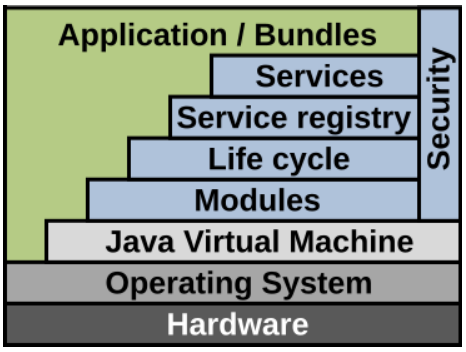

Figure 3: OSGi architecture.

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.

- Glassfish?
- Jetty?
- Servlet?

The project

- Everything happens in `Context.init`.
- in context.init hikariCP and liquibase db-migration tool.
- All database object access go through `DataManager` class. Like login (User). like a homebrew ORM (object relational mapper).
- `ObjectMapper` from Jackson library maps java objects <=> JSON.
- hikari manages connection pool database instances for example for PostGresql.
- usind direnv program to manage firebase admin environment variables. firebase admin requires a service key file which we pass to it using an environment variable containing the path.
- liquibase loads schema file from `changelog-master.xml`.

The final two pieces of the server are the object linking system that links devices with user, users with groups, geolocations, with devices etc…and the data manager system that sits between `Storage` and the processing in jax-rs resource methods.

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. [3]

[3] As defined in the Algorithms book.

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?

- 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?

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

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

```cpp
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.[4]

1. Sampling profiling.
2. Instrumentation profiling.

And for benchmarking

1. Micro benchmarking
2. Macro benchmarking

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

Algorithms

$$a = a + b$$

[4]Mathieu Ropert—youtube video