CS 331 Notes

Table Of Contents

Lecture 1

Key Topics of 331:

- Issues that arise in LLSD
- Teamwork
- Distributed Systems
- Web Servies
- Representational State Transfer (REST)
- Resource Representation
- Data Management
- ORM
- Transactions
- Middleware
- Software Architecture
- Architecture Tactics and Patterns

Lecture 1 Topics:

- What is Scale
- Factors that affect the scale of LSSD
- Life Cycles of SD
- Architecture (middleware)

What is Scale:
Scale is a multi-factored consideration when building software. It affects the team structure and composition alongside the design process and life cycle.
It is partially described by the project costs, required components and understanding of the problem description.
It is not a simple enough problem to be simply described by the cost of the program or a few simple measures.

Factors that affect scale / risks of LSSD:

Scope - Consider how much of the problem needs to be mapped correctly to the program, and reduce the scope for anything unneccesary.

Technology - Consider the program to decide which stack is neccesary to build an effective solution (consider the language (static / dynamic typing, etc.) and consider what the stack is optomised for.

Process - Consider the workflow process and how to split the program into subprograms that different groups / teammembers can work on seperately.

Resources - Consider the amount of resources to allocate and whether reallocation of resources may be necessary when building teams. Plan for these potential changes. Write code so that replacement can easily interpret and work with built code.

Security and Compliance - Ensure dangerous tasks are taken care of safely. Don't save credit card details and hash passwords. Use type safety to reduce runtimes.

Performance - Consider where performance gets the most effective remember premature optomisation is the root of all evil.
Consider continuous integration and deployment - auto track performance measures.

Budget - Consider how much budget exists, don't build over the top and waste more than you have.

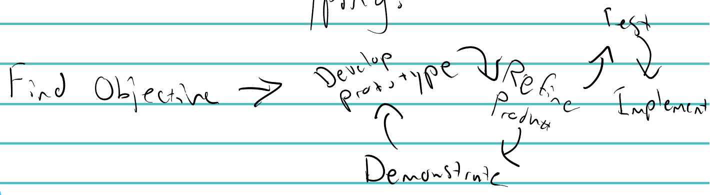
CS331 Notes

Software Development Life Cycle:

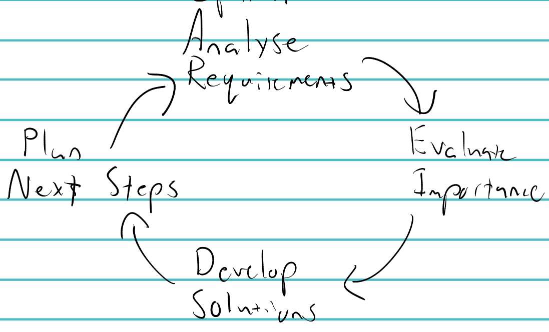
There are different Life Cycles for SD.
These include prototyping, spiral, waterfall and agile.
In class we consider Agile the best because it scales effectively. However for smaller projects it may not be as effective as there are a lot of unnessary procedures.

Agile Attempts to solve team-work issues.
To do so, they consider scrum masters who know how to communicate with the developers and explain this to the project management team. (loses direct communication, so can be very harmful).

Prototyping:



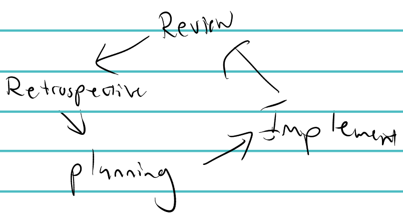
Spiral:



Waterfall:

$$\begin{tikzcd} \text{Int. Req.} \arrow[r] & \text{Design} \arrow[r] & \text{Implement} \arrow[r] & \text{Verify} \arrow[r] & \text{Maintain.} \end{tikzcd}$$

Scrum:



Domains of LSSD inlude:
- Enterprise Resources / Process Management (finance, supply chain etc.)
- Digital Customer Channels (Ecommerce Advertising etc.)
- Industrial Software (Equipment monitoring, Quality management, Analytics etc.)
- Smart connected products (Remote monitoring, Smart devices etc.)
- Learning networks (Digital Workspaces, eLearning etc.)
- Emergency Services (ERS management, Info Sys, Security, Survaillance etc.)

These often require Distributed Services (pg fill me in)
Client middle and service layers contrasting to standalone products.

What is middleware?
Middleware is software that allows different machines to communicate with each other easily. It adds abstractions that allow the user to use simple functions to solve problems that may be more complex / change if we were to directly speak to the other servers.
For example the conversion between Javascript and Java that has to happen whenever web servers communicate.
Often communicates this data using common protocols like SOAP, REST or JSON.

We want middleware because it allows us to have a standard package to ensure:
- Consistent Safety (not necessarily safe, but everywhere is equally safe).
- Reuse of Code leading to easy global modification if tasks are unsafe.
- less complex DEVXP because of simple functions to handle complex transactions.

Lecture 2 - Distributed Systems

Topics:
- What are Distributed Systems
- Network Infrastructure
    - Net Protocols, Internet Protocols, TCP TCP with Java
- Java Serialization

A distributed System is a system with multiple components located on different machines that are used to coordinate actions as a single coherent system to the end-user.
- This may include a desktop client, database server, cloud service etc.

Major problems in distributed systems include:
- Exactly-one delivery (where two systems send the same data, causing duplicate data).
- guaranteed order of messages (where data is sent from two systems at different times causing incorrectly ordered results).

Network Infrustructure problems:
- Sometimes computers and links fail independantly.
- Switches have finite space per packet.
- Links very in terms of capacity.
- Data can be corrupted.
- Incorrect delivery of packets.

Network Protocols:
We organise the protocols into layers that abstracts different components of the network. The application layer handles the interaction between clients and servers. It ensures that clients send correct types of information. It also defines what kind of information that can be expected. Finally it defines rules for when and how to send these messages.

Major use cases: data integrity, timing, guaranteeing throughput and security (though this can be handled on the transport layer.)

Transport Layer:
TCP provides reliable transport, flow control, congestion control and is connection-oriented.
It does not provide anything stated in the major use cases for network protocols.
UDP provides unreliable data transfer or any of the other uses. However this can be extended on the application layer to allow more custom control for more unusual programs

Network Layer provides a packet delivery service between different host machines.
This allows different machines to communicate and ensures the data is sent to the correct device.

The layers depend on the interface of the lower levels, not the implementation.

An IP is the way the network layer distingushes between different host devices. We send a packet from one device to another sending the source IP address, destination IP address and a payload of the data we want to send to the destination.

TCP (transmission control protocol) is a transport protocol that creates a virtual connection between a pair of processes.
It allows the application layer above it the ability to send data without worrying about:

- the limited message sizes of a packet.
- the destination of the data (i.e. ports and IP addresses)
- the chance of data being lost / corrupted
- message duplication
- flow control

One way it could do this is with a log, timer, by sending acknowledgement packets with each packet, a sequence number, a checksum and a buffer.
To handle message sizes, it can keep a buffer of all sending information and break it down into payload sizes and send them one at a time. To handle corrupt data, it can send use a checksum that is invalid if data was corrupted, and the other recipient can send back a false acknowledgement packet to tell the host machine to resend the data.
To handle lost data, it can resend the packet if it takes to long.
To handle flow control, it can use the sequence number to ensure that even if the data is recieved in the wrong order, the recipient can check the sequence number and wait for any missing data.

A socket is bound to an IP address and a port number and it identifies where the application should send data to, and recieve data from.
A socket converts the data to be sent to be something the internet can interpret.
To allow this to happen, we need to describe each class as serializable, as this forces the compiler to decide on how to serialize it into a primitive datatype.

The java serializer describes what an object looks like, this allows the client to rebuild each object by deserializing it.
When an object is serialized, it becomes a tuple of features consisting of:
the class name, it's version number, it's handle, the number of instance variables it has, the name of each variable and the type of each variable.

Then we also send a set of objects that consist of the list of the class handle, followed by each instance variable for each class.

Serializing data allows us to send data to the network easily, save it to a file and make a deep copy in memory. Transient data will not be sent.

Lecture 3 - Web Services

Topics:

- What kinds of middleware are there?
- Service-oriented architecture (SOA)
- HTTP
- Servlets / servlet containers
- Web Services and REST

Types of Middleware:
Socket Programming Middleware:
Socket programming based middleware is a type of value-driven middleware:
It only abstracts away the TCP component, so is very low-level as the developer has to work with streams manually and pass the values directly to through the stream.

Remote Method Invocation Middleware:
This has abstracted away the socket, and instead the client commicates with a proxy for the server. This allows the programmer to avoid directly handling streams and communication occurs through invoking object proxies and methods.
This works by having an interface that defines how the client can interact with the server and an interface for how the server can interact with the server. Then it calls an implementation of that interface that handles the socket-programming (value-driven.)

Service-Oriented Middleware:
This uses a series of services that communicate with a defined format that is not necessarily directly passing the objects. This allows for the client and server to communicate in different languages.
For example, the server could work in Java, but the client might only understand javascript, to work around this, they could communicate using JSON, then interpret what the other says into a more useful format.
To do this, we need to define a serialization that allows us to marshal / demarshal the problem into JSON and back from JSON respectively.
We communicate by invoking requests in JSON or XML because we cannot directly call methods as we don't know the language we are calling to / from.

It is very useful when a lot of services have vastly different expectations (programming languages, data types, information about data etc.)
however it's not that effective for performance or cost (because of loss of potential gains from not needing some part of the abstraction sometimes [such as services using the same language].)

Look further into architectures with: https://www.developertoarchitect.com/lessons/

HTTP Protocol:
uses two kinds of messages: requests and responses
A request is constructed with the method, the url, the version, a header and a body.
A reply is a version, a status, a reason, a header and a body.
A url has http://serverName[:port] [/pathName] [?query].

Servlets:
Servlets are middleware that abstract away the http requests and sessions. They control how the program works based on the model and the view.
Servlets are singletons that handle each server. If a servlet gets multiple threads of requests, it needs to handle them one at a time for thread safety.
A servlet container manages the different servlets and the lifecycle of the servlets.

Lifecycle:
the servlet is first initialised, then performs it's serve for each request it's given using the serve() method. When the servlet container is shutting down / resources need to be freed, the servlet is destroyed and the destroy() method is ran which frees the data the servlet is holding and backs up any potentially dangerous problems.

The HTTPServlet class is an abstract class that has doGet, doDelete, doPost and doPut methods that handle the different kinds of interactions a user will have with the server.
To make our own servlet, we will override these methods to allow users to get results.

These methods take in a Request and Response object (we do not call these methods directly, so we don't need to worry about that.) but these allow us to write using the resp.getWriter() method.

When doing this, we also need to make a web.xml file that tells the program the structure of the file.

REST:
REST stands for Representational State Transfer, was designed by Roy Fielding.
There are 5 main principals of REST:

- Resources need to be addressable
- Uses a uniform, constrained interface
- Focused on representing the problem
- Communicate Statelessly
- Hypermedia As The Engine Of Application State (HATEOAS)

Addressable resources means that any resource should be referenced by a unique address that is completely self-contained. Doing this allows us to send small amounts of information to express complex objects (i.e images) and - if the other user has the data they don't need to download the entire thing.
REST often uses URI's to represent the various resources.

Uniform Constrained Interface means that the interface is used in the same way everywhere and is only used for it's intended purposes. (I.E using the right HTTP methods.)
This also allows for caching of repeated uses of an interface method.

Stateless means that there is no session maintained over multiple visits, instead we store this data as a cookie. We do this so that we can easily control what operations are valid to allow data changes to be atomic and isolated.
We also focus on using a uniform constrained interface to ensure that transactions are consistent and durable.

In REST we focus on being representation oriented with communication. This means we communicate between the server and the client using MIME types. These are generic representations of data like text/plain, text/html, application/xml and application/json. This allows consumers to negotiate for datatypes without adding a lot of extra steps to generate thousands of different layouts of data. (universal few datatypes).

We also use stateless communication so that the server can change without influencing the affects on the client (as the other server might not have the same state as the original server after changing.

Finally we consider HATEOAS [hypermedia as the engine of application state]. This means we should allow user traversal with hypermedia links as opposed to getting users to find the links themselves.

This allows the server to track where the user may go next and cache idealistic data?

HTTP Methods:
GET is used when the client requests something from the server.
HEAD is used when the client wants the metadata.
POST is used for processing form data.
PUT is used to update some data on the serverside.
DELETE is used to remove a resource on serverside.
OPTIONS us used to request the available options from serverside.
TRACE is used to track what requests happened.
1xx responses are informational
2xx response codes are successful
3xx response codes are redirection
4xx response codes are client errors.
5xx response codes are server errors.

HTTP is very useful because of it's stateless architecture (and controlled state cookies)
It is also very useful because it has negotiable content, that is that a client can request different kinds of content types and the server will return the best one that it can for the browser.

Lecture 4

Topics:

- JAX-RS
    - Programming Model with JAX
    - REST with JAX
- Example (how to code in JAX)
- JAX API

JAX-RS is a specification for frameworks of interfaces to simplify web-development.
Key Features:

- Annotation Based
- Parameter Injection
- Marshallers / Unmarshallers

We use the RESTEasy implementation of JAX-RS.
The JAX_RS implementation provides a servlet class HTTPServletDispatcher that delegates processing of HTTP requests to different Application Services.
From a request's perspective, it is sent to a servlet container that reads it's metadata to find the servlet to use which is then sent to the application to be processed. A response is generated here which is sent back through this tunnel.

This allows the servlet to provide both a preprocessing for the request and post-processing for the response.

To use a servlet, we need to:

- create a Resource with methods to process requests.
- create an Application class that overrides the getSingletons() method.

JAX-RS creates an instance of every application subclass and runs their getSingletons() method.

Application Implementation Details:

- create a path for the application with @ApplicationPath("/{path_name}")
- make the application extend Application
- make sure the constructor adds all required resources to the singletons.
- in the getSingletons() method return the set of all used singletons.

Example:

```
@ApplicationPath("/services")
public class HelloApplication extends Application {
        private final Set<Object> singletons = new HashSet<>();

        public HelloApplication() {
                singletons.add(new GreetingsResource());
        }

        public Set<Object> getSingletons() {
                return singletons;
        }
}
```

Path lookup based

service resource needed to build application

⋮

more resourcuses may be needed.

Resource Implementation Details:

- create a path for the used resource. (@Path("/{path}")

for every method that gets or requests some data:

  - tell the method the request type (@GET, @POST etc.)
  - create a path to the method. @path("path")
  - tell the program what kind of result it gets (@Produces(DataType)) s.a. JSON.
  - tell the program what kind of data is uses (@Consumes(DataType)) s.a. JSON.

You can set paramaters as queryParam using @QueryParam("{param}") dt name.
You can set defaults of these params using @DefaultValue("{value}").

Example:

```
@Path("/greetings")   services/greetings
public class GreeetingsResource {

        @GET            returns a resource
        @Path("hello")   services/greetings/hello
        @Produces(MediaType.APPLICATION_JSON)   Produces a json item.
        public Resource sayHello (
                    @DefaultValue("Human") @QueryParam("name") String name
        ) {
                String json = "{ \"greeting\": \"Hello, " + name + "!/" }";
                return Response.ok(json).build();
        }
}
```

services/greetings/hello?name=Jeff
would use name="Jeff"
no query would return "Human"

success   response body

To request this data in Java, we would create a new client, then the client will target http://.../services/greetings/hello?name=bob, request it, and call get.
Then we could read this by using response.readEntity(String.class) and print it and close the client.

e.g:

```
Client client = ClientBuilder.newClient();
Response respoinse = client.target({url}).request().get();
String json = response.readEntity(String.class);
System.out.println("Response JSON: " + json);
client.close();
```

In javascript, we can use this resource with:

```
const input = document.querySelector("#txt-name");   Input field like textbox
const name = input.value;
const response = await fetch("./services/greetings/hello?name=${name}");   (say bob entered)
const json = await response.json();
const output = document.querySelector("#span-greeting");
output.innerText = json.greeting;
```

→ output field / plaintext etc.

shows Greeting: Hello, bob in dialogue box.

Consider if we wanted to make a Parolee program that allows us to:

- update a parolee's location, query for this location, query the database for a set of parolees, "maintain the database", update and remove parolees.

To do this with JAX-RS, first we consider the different operations we need to do.
These will include:

- get a list of all parolees with	/parolees
- get information about parolee movements with	/parolees/{id}/movements.
- get data about a parolee with	/parolees/{id}
- remove a parolee with	/parolees/{id}
- update a parolee with	/parolees/{id}

This appears to have duplicate ideas, but because the requests are different CRUD operations this works.

potentially we might want to get a small portion of the parolees, this could be done with parameter injection and using GET /parolees ? start=0 & size = 10 and using start, size as paramaters with the @QueryParam feature.

JAX-RS automatically converts strings to data types. this is done directly for primitive types
If there's a constructor with a single string object, it uses that.
valueOf() can also be used *[?]
If it's a collection of generics that satisfy one of these it returns an arraylist of them.

If a request succeeds you should return either a 204 with No Content as a response or a 200 Ok with a non-null response.

If a URI doesn't exist you should return a 404 Not Found error.
If a client attempts to use an incorrect HTTP method return a 405 "No Method" response.
If a client request an unsupported content type return a 405 "Not Acceptable" response.

Exceptions can be handled either: directly by JAX (if WebApplicationException)
or by an ExceptionMapper if one is registered for the specific exception. #notshown

A ClientBuilder builds Client s that target a WebTarget that can be used to request an InvocationBuilder that inherits a SyncInvoker.

A SyncInvoker may throw ProcessingExceptions or WebApplicationExceptions, so a client needs to catch these exceptions. (can use try-with-resource syntax). [if uncaught could cause resource leaks.]

Lecture 5:
Topics:
- REST Representation Orientation
- Resource Representation in JSON with Jackson

Marshalling:
A marshaller converts Java Objects into HTTP messages to be sent to the client. In JAX-RS, a MessageBodyWriter handles this.
An unmarshaller converts HTTP messages to Java Objects. In JAX-RS, a MessageBodyReader handles this.
typically we need both for each in/output type. At runtime the appropriate one is used for an accepted MIME type, this uses the @Produces() and @Consumes() annotations.
The ReaderAndWriter needs to be added to the classes hashset of the application.
this uses the "application/java-serialization" MIME type.

Registering Marshaller on Client:
To do this, simply use Client.register(ReaderWriter.class).
To request a java-serialized object, you need to use accept(app/java-ser) before using get().
To send a response, you need to use .post(Entity.entity(playload, app/java-ser)).

Jackson is Java's JSON program, it has it's own Marshaller it expects:
- classes to follow JavaBeans (serializable class, no-arg const, priv vars, pub get/sets)

This is because it will set each property seperately (and inductively for lists / object props.)
- The getters / setters need to have @JsonGetter("propName") / JsonSetter("propName") annotations, we can ignore it with @JsonIgnore.

Lists can be marshalled, however it will always return an ArrayList.
Maps become JSON objects. This becomes an object of entires, each entry is an object that has a key mapping to it's object.

If an object isn't JavaBeans we need to make a custom Serializer / Deserializer that overrides the Jackson one.

Jackson maps Keys to strings by calling toString(). To Deserialize them, we need to use the @JsonDeserialize(keyUsing = key_deserializer_class.class)

When martialling, we can have duplicate items that are not recognized as duplicates, causing duplicate objects.
To fix this we can use @JsonIdentifyInfo(generator: ObjectIdGenerator: property: string) then reference the id in each object. This is then martialled / demartialled and uses the same object if it has the same id. This also handles cyclic references.
The cyclic references can also be done by using @JsonIgnore to ignore one of the ref in cyc

Sometimes we need to specify the subclass, to do this, we need to use the @JsonTypeInfo(use = className, property = "type")) [default use = JsonTypeInfo.Id.CLASS
otherwise use JsonTypeInfo.Id.NAME and define @JsonSubTypes(@JsonSubTypes.Type(value = Cat.class, name = "cat"), ...)
the first generates the "type" : "path.path...type" the second generates "type: "name".

Lecture 6:
Topics:
- Distributed Data Architecture
- OO-relational paradigm mismatch
- ORM with JPA (Java Persistance API)

CAP Theorem:
- tradeoff between consistency, availability and partitionability.
- Consistency means all clients see the same data regardless of the server dispatcher.
- Availability means how capable a server is to respond to any request at any time.
- Partition tolerence means how well a system works even under a network failure.

Example, bank with 2 atm's network
can deposit, withdraw or check balance
balance >= 0
stored balance on both atm's - no central db
balance is updated on both atm's over network if a user does something
if atm1 turns off, when someone uses atm2 it may refuse to ensure consistency
otherwise atm2 may be used, but can cause consistency mismatches allowing the user to withdraw more than they have. (resulting in negative balance)
In this case, consistency may be preferred over availability.

Example 2: commenting on a social network.
Can leave comments, veiw other people's comments
comments are stored on a database but because of network flow it takes time for it to be updated.
We could focus on availability to allow users to always comment, but some comments may be hidden.
We could also focus on consistency and not allow users to comment until all comments are shared with all servers, but some users may not be allowed to comment.
In this case availability may be preffered over consistency because missing comments is not a big deal.

Notice we don't need to be completely one-sided, it may really annoy customers that they frequently cannot withdraw money from the ATM costing the bank significant funds. To fix this it may be a good idea to increase availability by allowing users to make small withdrawals / deposits that are unlikely to put the bank at a significant loss. However this would still have the potential to allow users get inconsistent funds and negative balances.

Methods of Maintaining Consistency:
Active Replication:
Clients send requests to all replicas. Atomic multicasts and middleware ensure order and atomicity of requests so that if sime requests are made, each server does each request in the same order.
They also ensure that if any server does the request any - non-crashed - server will also process the request. (Using this we could potentially update crashed servers once they come online any only allow usage after the server is fully updated).

[Notice unavailiability with multiple copies is not really unavailable if we can still access at least one other copy, but in terms of CAP, it is still unavailable!]

Master-Master Replication:

- Each server communicates it's changes with each other server randomly. Allows for slight inconsistencies in data, but generally all data is in most servers.

DNS is the Internet's lookup service for IP addresses, focuses on scalability and availability. To meet this, DNS has been designed using replication, partitioning and caching.

The DNS is partitioned into different layers, the root layer that references the TLD server layer (has the .org, .com, .edu etc. lists) that contain the Authoritative server layer (for example .com has yahoo.com, google.com etc.) There are several different replicas of these TLD servers.

When a client requests a server's page, they first need to find the IP address, to do this, they request their Local DNS server, This accesses the Root DNS server to find which TLD to use, it then looks in the TLD DNS server where it then may need to look in other DNS servers (i.e. if the DN is www.auckland.ac.nz it would need to look in the nz TLD DNS, then the .ac.nz DNS server, then auckland.ac.nz Authoritative DNS which will have the ip for the requested item.

Servers will also cache recently used and frequently used webpages. DNS servers use a TTL (time-to-live) model to ensure stale (unused) cached data is replaced with new data. They cache the IP addresses of other DNS servers they discover when processing queries. If the TTL is very long it's possible that the Cache is outdated, reducing consistency
If the TTL is very short it's possible that a DSN / Webserver is not stored in cache when we want it, reducing availability.

Paradigm Mismatch:
Objects cannot be simply mapped to Relational Databases because of small differences in:

- Granulity: It may not be preferrable to have 1:1 table:class mapping.
- Subtyping: Hard to create tables for inheritance heirarchies. This causes polymorphic queries and associations to be challenging.
- Identity: hard to tell if two objects are the 'same' because references from a DB don't get stored at the same memory location. Granulity joins here where a row may be representing multiple objects here.
- Associations: OO associations use references to other objects in memory, RDB associations use foreign keys. [?When joining there is no directionality in RDB's?] Many-Many associations are complex in RDB's. (association classes help)
- Data Navigations: OO can keep easy references in memory, however RDB's do not directly store all query tables. This means that chains of data collection methods are very fast in OO, but very slow in RDB's. The reverse is that RDB's are fast when handling.
  This is because it's much faster querying for all the data at once rather than making a lot of smaller queries in RDB's, but in OO we can store what we query to not have any extra overhead when we do this.

We want to aviod persistance leaks into the Domain model classes to avoid potentially mistesting persistance with (destructive) domain model persistance testing.
This may appear slow, because we always need to retrieve from a db, but it's only for the higher level details and the program can store all used objects outside the domainModel.

ORM's are programs that allow us to map a domainModel to a RDB. JPA is a specification for an ORM, and Hibernate is a implementation that follows JPA for the H2 database.

We want to define the ORM s.t. we an use all required CRUD operations, but to change the database software we wouldn't need to modify the domainmodel.

To deal with this, we can use annotations (decorator pattern) to allow the technology to choose how to implement similar features.

Therefore we need to have annotations that relate the different tables and components to tables. To do this, an @Entity is a table and an @Id is an id for the table.

JPA aims to provide transparent and automated persistance (SOC and abstracts out persistance details)
Persistance classes must follow Java Beans. Collections must be typed to an interface.

DTO's are classes that only get the required information from the persistance storage.

Lecture 7:
Topics:

- JPA / Hibernate Entity vs Values
- Mapping persistant classes
- Mapping Collections
- Mapping entity associations

An entity has a database identity, so it can be independantly queried. This means it has it's own lifecycle.
A value type is part of another object, and thus doesn't have it's own lifecycle. Therefore we should never share value-types.
If something should only exist with another object, it should be a value-type, otherwise it should be an entity.

Entity classes need to be annotated with @Entity.
Entity classes must have a database identity field annotated with @Id.
@GeneratedValue allows the Id to be constructed when a new object is created.
Value-type classes need the @Embedded annotation, this annotation causes the properties of the value-type to be columns in the entity table it came from.
@Transient is not mapped to the table
enums are ordinal unless using the annotation of @Enumerated where they are the string.
@Temporal(TemporalType.{DATE|TIME|TIMESTAMP}) maps to a date-time column.