**2-7**

For Capstone:

Need SRS with summary, functional reqs, non-functional reqs, db schema, workflow diagram

* Home page
* Registration page
* Login page
* Contact Us
* A does something page
* DB needs Create, Read, Update, Delete
* JavaScript is only form validation

SDLC: Waterfall

* Requirements Analysis
* Design
* Develop; Code
* Test
* Go Live
* Maintenance

SME = Subject Mater Expert

SRS = System Requirement Specification

FRD = Functional Requirement Document

Non-FRD i.e., hardware, licensing, network; setting up the environment; support or facilitate the FRD

RTM – Requirements Testing Matrix

**2-8**

Agile – set of values and principles; BA now called Project Owner

* **Individuals and interactions** *over* processes and tools
* **Working software** *over* comprehensive documentation
* **Customer collaboration** *over* contract negotiation
* **Responding to change** *over* following a plan

CI&CD – Continuous Integration, Continuous Development

Scrum:

* Framework which is intended to be adaptable to the team’s needs
* Iterative, focused on small deliverables
* Agile
* Self-managed by team
* Three Pillars
  + Transparency
  + Inspection
  + Adaptation
* Values
  + Commitment
  + Focus
  + Openness
  + Respect
  + Courage
* Roles
  + ScrumMaster – “Coach” of the team, addresses roadblocks/problems; Burn-down report
  + Product Owner – Represents business, often the Business Analyst; “Grooms” user stories
  + Development Team – Creates plans for Sprint & Sprint Backlog
* Sprint:
  + the working hours of the team, 2-4 weeks
  + Each iteration of the Scrum
  + Designed to produce a single potentially deployable feature
  + No requirements changes
* Artifacts of Scrum (documents)
  + Product Backlog = Requirements Document; holds User Stories and Epics
  + Sprint Backlog = the User Stories being worked on in the sprint
  + Increment = body of work accepted by product owner at end of sprint
  + Additional: Report / Result of Increment
* Events
  + The Sprint – heartbeat of Scrum
  + Sprint Planning Meeting – involves everyone, not more than 8 hours but should be 4 or less
  + Daily Stand-up Meeting / Daily Scrum – Dev Team, everyone else optional; Not more than 15 minutes; Three Questions:
    - What you did yesterday
    - What you are doing today
    - Blockers
  + Sprint Review – Everyone and Stakeholders – demonstrate product and seek acceptance
  + Retrospective Meeting – everyone – how to improve process; 3 hours max for one month Sprint and less for less; Scrum Master and Dev Team create report for Stakeholder (shows Velocity)
* Product Backlog Items
  + Epic – big chunk of work; feature, customer request, or business requirement
  + User Stories (medium chunk of work XD)
  + Task – small chunk of work
* User Stories
  + Describe type of user and what they want and why
  + AS A <role>, I WANT TO <action>, SO THAT <reason>

**2-9**

* Velocity = the number of Story Points you can complete in one Sprint
* Out of the of the Sprint R

**2-10**

DBMS

* System Software
* Create and manage DB
* Systematic way to create, retrieve, update, & manage data
* Manages
  + Data
  + Database Engine allows data to be accessed, locked, and modified
  + DB Schema defines db’s logical structure
* 3 Foundational elements
  + Concurrency
  + Security
  + Data Integrity
  + Uniform admin procedures
* RDBMS
  + Table
  + Row
  + Attribute / Column
  + Primary Key – uniquely identify each row
  + Composite key – primary key with multiple attributes
  + Relational Schema – structure of the relation
* ACID Properties – scoped to a transaction, unit of work defined by programmer; can combine one or more DB operation
  + Atomicity – define a transaction that contains UPDATE, INSERT, and DELETE, all treated as single unit; ether all change the DB or none do
  + Consistency – never leaves the DB in half-finished state – i.e. Docebo – delete user = delete all posts; cascading deletes
  + Isolation – transactions separated until finished
  + Duraility – db will track pending changes in such a way that the server can recover from an abnormal termination; use a log to store uncommitted transactions
* Data Modeling – analysis and design method (the plan for the DB)
  + Define and analyze data requirements
  + Define logical and physical structure that support requirements
  + Refers to the practice of documenting software and business system design
  + Three types
    - Hierarchical
    - Network
    - Relational
  + 3 levels
    - Conceptual representation of system, composition of concepts used to know, understand, or simulate; brainstorming; highest level relationship between entities
    - Logical – describes data with as much detail without regard to how to implement
      * All entites and relationships
      * All attributes for each entity (ALL DB SHOULD HAVE startDate & updateDate – best practice)
      * Primary keys
      * Foreign keys
      * Normalization– Process or set of guidelines to optimize DB; reducing redundancies; guidelines = normal forms
    - Physical

SQL – Structured Querying Language

* DDL – Data Definition Language
* DML - Data Modification Language
* DQL - Data Query Language
* TCL - Language
* DCL - Data Language
* SQL evaluates: From>Where>Group By>Having>Select>Distinct>Order By>Limit
* Keywords
  + **Select**
  + **From**
  + **Where**
  + **Order** **By**
  + **In**
  + **AS**
  + **Distinct**

**2-11**

* Keywords continued
  + **Insert** **Into** <tablename> (col1, col2, …) **Values** (<data>, <data>, …), (<data>, <data>, …),..
    - Does not support alias
  + Where <row> **Between** <data> AND <data>
  + **NOT**
  + Migrating **INSERT INTO** table2 **SELECT** \* **FROM** table1 **WHERE** <row><operator> data
  + **UPDATE** <table> **SET** <row>=<data> **WHERE** <row>=<data2>
  + **DELETE** <table> **FROM** <row>=<data> **WHERE** <row>=<data2> (not used in production; if deleting try to put in a transaction {see below})
  + **START** **TRANSACTION**; …[one or more DML statement]; **ROLLBACK**; (or **COMMIT**;)
  + **TRUNCATE** (a delete that cannot be rolled back)
  + **GROUP BY**
  + **HAVING** <group condition>
    - Order of DQL: SELECT\*-FROM\*-WHERE-GROUP BY-HAVING-ORDER BY-LIMIT
  + Aggregate (built in) functions examples
    - **SUM**
    - **AVG**
    - **MAX**
    - **MIN**
    - **COUNT**
    - **YEAR(‘<year>’)**
  + **LIKE** – Use with % for multiple wildcard and \_ for single character
  + **SHOW, USE, DESCRIBE** – needed for command line
  + **LIMIT** used to constrain the number of rows

**2-14**

* Aggregate Functions
  + String Functions
  + Numeric/Math Functions
  + Advanced Functions
  + Date Functions
* ETL
  + Extract
  + Transform
  + Load
* SQL DDL
  + CREATE
  + ALTER
  + DROP

**2-15**

* Constraints – four types
  + Entity
    - Primary Key
    - Unique
  + Domain
    - Data types
    - Not Null
    - Defaults
  + Referential
    - Foreign Key
  + User-defined
    - Check
    - Triggers
* Joins – FROM <left table> JOIN <right table> ON left.field = right.field
  + Cross- matches all data from left table to right table; Cartesian product; Used when no common data
  + Inner – returns common data; Can use USING instead of ON if named the same in each table
  + Natural – joins based on fields with the same name (don’t need to use ON)
  + Left Outer – gets all data from left table and all common data from right table
  + Right Outer – gets all data from right table and all common data from left table
  + Full Outer - Union
  + Self – table is relating to itself – just use JOIN

**2-17**

GITHUB

* Version control – manage changes
* [C:\Users\grend\Downloads\Hands on Activity Git and GitHub.docx](file:///C:\Users\grend\AppData\Roaming\Downloads\Hands%20on%20Activity%20Git%20and%20GitHub.docx)
* Share between github accounts: Fork

**2-18**

Java

* Project needs at least one package with at least one class with at least one method
* Main(String[] args) method is the entry point into the program
* Package name all lowercase
* Extension of Java classes is .java
* Public static void main(String[] args)
  + Intellij type psvm the TAB to get main boilerplate
  + <ctrl+j> to get list of boilerplates
* Classes in upper camel case
* Methods in lower camel case
* Variables lower camel case
* Constants all upper case
* Interfaces upper camel case, can put I in front of interface name (IComparable)

Type casting

* Widing casting (Auto type conversion) -> convert a smaller type to a large type
* Narrowing (Explicit type conversion) -> convert a larger type to a smaller type

**2-22**

Scanner object

* Used to read data from Console, file, etc
* Can take data from all types (files, inputstreams, int, etc)

**2-23**

Loops

* For loop
  + for(int i = 0; [expression]; i++) {[code]}
* While loop
  + while([expression]) {[code]}
* Do-while loop
  + do{[code]} while([expression])

**2-24**

OOP

* Four Pillars
  + Encapsulation
  + Inheritance
  + Polymorphism or Overriding
  + Abstraction or Overloading
* Access Modifiers
  + Public – accessible from anywhere
  + Private – accessible to class only
  + Default -accessible to class and other classes in the package
  + Protected – accessible to class and subclasses
* Classes
  + Revolve around real life entities
  + Objects share common properties and methods
  + When defined
    - Fields – class level and local variables; Class values automatically declared, local must be declared
    - Geters/setters – methods to get or set variables
    - Methods
    - Constructors
  + Objects are an instance of the class
    - **Instantiation**: Instantiate == create an instance == create an object of a class
  + By default, there is a constructor with no arguments if there are no constructors in the class
  + Pojos or entity class contains only variables and getters/setters
  + **Overloading**
    - can create methods with the same name
    - must have different arguments (number or datatypes)
  + **Encapsulation**
    - Data Hiding variables from other classes – Class level variable always private or protected (if parent class)
    - Binds data to functions that manipulate data (constructor, getters and setters)
  + **Polymorphism**
    - The ability of a variable, methods, or object to take on many forms
    - Uses inheritance to use the variables and methods of other classes
    - **Upcasting** – make a parent object with child constructor
    - **Downcasting** –
  + **Abstraction**
    - **Is about hiding unwanted details while showing most essential information**
    - Separating the implementation from the declaration
    - Implemented through abstract methods and classes and interfaces
* Methods
  + Named block of code
  + Invoked by name
  + Optionally have parameters and/or return type
  + Unique signature-based name and parameters
* **REMEMBER WHAT STATIC MEANS**
* Inheritance
  + Represents the IS-A relationship which is also known as parent-child relationship
  + You can reuse parent variables and methods in the child class

**2-25**

* Abstract Classes
  + Can have abstract and concrete methods
  + Can have variables (variables cannot be abstract) (not public, static, final by default)
  + All abstract methods must be overridden by child classes
  + Cannot be instantiated
* Interfaces
  + Only abstract methods and final variables
    - Don’t need keywords
  + **Methods are public and abstract**
  + **Variables are public, static, final by default (or implicitly)**
  + Cannot be instantiated
  + Cannot have constructor
  + Is *implemented* by another class
  + Uses keywords interface to create and implements to use
  + Cannot extend any class
* Enums
  + Contains a set of constant values (final, public, static)
  + Cannot be created inside a method
  + Can be created outside of a class
  + When referenced the datatype is the name of the enum
* Generics
  + Creates a single class, interface, and method used with different types of data objects

**2-28**

Exceptions

* Checked also called IOExceptions
  + **A checked exception forces you to use either try-catch block or throws keyword**
  + Classes are attached with some possible Exceptions
  + Are compile time exceptions
* Unchecked also called Runtime Exceptions due to programmer mistake
* Try-catch
  + Only one Try-catch per method
  + Try {//code} catch ([exception type]) {//code}catch ([another exception]) {//code} …
  + Finally
    - Optional
    - Always runs whether there is an exception of not
    - Only one per try-catch
* Throw
  + keyword used to manually/explicitly throw an exception
  + can only throw one
  + Usually used to theow custom exceptions
* Custom Exception
  + Save time with easily understood naming

**3-1**

**3-2**

Java Collections Framework

* Library of advanced data structures
* Primary interfaces
  + Collection
    - Adding/removing elements or clearing
    - Iterating
    - Testing contains
    - All are generic types; can pass in any datatype
    - All in java.util package
  + Map
    - Missed
* List Interface
  + ArrayList
    - Implements dynamic array
    - Implements collection<E>
      * Collection<E> extends Iterable<E>
    - Resizeable
    - Elements can be added and removed
    - Allows dups and nulls
  + LinkedList, Vector, Stack – classes to use

**3-3**

Tree is a recursive data structure where each node can have 2 children at most

Hashing is a technique that is used to uniquely identiy a specific object from a group o similar objects. Hash values are dependent on the JVM so they can change every time it is run. Output will end up the same because it is the same data computed with whatever JVM value they use. Hashes aren’t meant to be sorted.

Map interface defines a mapping of key values. Keys must be unique, associated with a single value. Key type is not mutable

**3-4**

* StringBuffer
* StringJoiner
  + New StringJoiner(“,”) – comma delimited
  + New StringJoiner(“,”, “[“, “]”) – [str,sstr]

Functional Interfaces

* Functional interface has a single abstract method called the functional method
* Interface has exactly one abstract method
* Interface implements one behavior
* May be other default and static methods but only one abstract
* Default methods have implementation
* **Free to add as many default and static as you like**
* Can specify Object’s class public methods
* Known as SAM (Single Abstract Method)
* Java.util.function
* @FunctionalInterface – forces it to only have 1 abstract method, but is not required
  + Can declare an abstract method that overrides a method from java.lang.Object that doesn’t count towards the rule of only one abstract method
* **Can instantiate Functional Interface but do so anonymously**
* **Represented using lambda expressions, method reference and constructor references**

Lambda Functions – anonymous functions implement Functional Interface

* Written where they are needed
* Used to define inline implementation of functional interface
* Only has body and parameters
* Compiler infers the return type for us
* Syntax
  + (param\_list) -> {function\_body: define logic}
    - Param\_list can be empty; need to declare type
    - Body can have 0 or more statements
    - Optional return type – compiler automatically returns the value if the body has a single expression to return
    - {} needed if there is more than one statement or if there is a return

Concurrency

* Is the ability to do multiple computations at the same time
* Can be achieved by distributing over multiple computers, processors, cores
* Execution environment made by OS
* Has private resources
* Communication is possible but difficult
* Multithreading is a Java feature that always concurrent execution of two or more parts of a program for maximum utilization of CPU – each part called a THREAD
* A thread, in Java, is a path followed when executing a program
* All java programs have a thread, main thread, created by JVM
* Threads can live in a process – can be created and stopped dynamically
* Java creators have designed two ways of creating treads, implementing interfaces & extending a class
* Process can have multiple threads; when process starts it is given memory and resources shared by all threads
* Thread info
  + Id – unique identifier
  + Name – given name to refer to thread
  + Priority – helps JVM decide execution order
  + State
    - NEW
    - RUNNABLE
    - BLOCKED
    - WAITING
    - TIMED\_WAITING
    - TERMINATED
  + threadGroupName – where the thread belongs to be able to managed in groups

**3-7**

JDBC

* Steps
  + Register driver
  + Initialize
    - Database URL
    - Username
    - Password
  + Prepare Statemen
* Send Statement
* Get Results
* Separate class for queries

**3-8**

JDBC with OOP

* Start with abstract DAO
  + Register driver
  + Connect to DB
  + Safely close
* Create interface for each DAO
  + Contains queries
  + Creates structure for methods
* Implement DAO
  + Write logic for interface methods
  + Start with register/connect method
  + End with close method

**3-11**

* ORM (Object Relational Mapping – Translate programming to database
  + Class 🡨🡪 Table
  + Java Object 🡨🡪 Row
  + Class Attribute 🡨🡪 Column
* JPA
  + Java ORM
  + Package in Java
    - Better to use getters/setters rather than constructors
  + Implementations include Hibernate, EclipseLink, & Apache OpenJPA and more
  + JPA permits developer to work with objects rather than SQL – persistence provider
  + EE standard ORM (EE is used for making online applications)
  + JPA API is a set of rules and framework to set interfaces for implementing ORM
* Hibernate
  + Implementation of JPA API
    - Has more classes on top of JPA
    - Java classes to db tables
    - Java data types to SQL data types
    - PERSISTENCE
    - Hibernate Validator used to validate Forms
    - JDBC not OO so Hibernate makes it OO by representing record as Object
    - Using straight Hibernate locks into implementation since others would use other methods so no ORM switching (not as useful – restricts JOA)
    - Hibernate.cfg.xml used to map POJOs to DB
    - Persistent object
      * Default constructor
      * ID maps to primary key
      * Class variables private and getters/setters defined in JavaBean style
    - Configuration
      * Class which is present in org.hibernate.cfg
      * Crated once
      * This is the mapping (hibernate.cfg.xml)
      * Two components
        + Database Connection
        + Class mapping
        + Configuration cfg = new Configuration(); //activates framework
        + Cfg.configure(); // read both cfg file and mapping files
      * Session Factory
        + Interface present in hibernate package
        + Thread-safe object and used by all threads
        + Created during app start and kept
        + Need one per configuration file
        + Multiple dbs = multiple SessionFactory objects
        + SessionFactory factory = cfg.buildSessionFactory(); //gathers meta-data in cfg. From cfg it takes the JDBC info and create connection
      * Session
        + Used to get connection with db
        + Designed to be instantiated each time an interaction is needed
        + Session objects should not be kept open long; created and destroyed as needed
        + Session used to preform all CRUD operations
        + Session session = factory.buildSession();
        + Session methods used to manipulate data
      * Transaction
        + Enables you to achieve data consistency
        + Transation tx = session.beginTransaction();  
          tx.commit();
        + Methods

Begin()

Commit()

Rollback

setTimeout

* + - * Hibernate App Stages
        + Phase 1

Clesses – creating entites

DAO/Services Classes

Create, Update, Delete, Find

* + - * + Phase 2

Mapping File: Hibernate Configuration

Configurations – db connection inf, scheme level settings, entity mapping

* + - * + Setting up Hibernate

Cret Maven project

Configure Maven Dependencies for Hibernate and DB in pom.xml

* + - * Annotations
        + @Entity – maps class to DB table
        + @Table(name=<table name> – used if class name is different from table
        + @Id – maps a field to a primary key
        + @Basic – tells JPA that variable is regular attribute
        + @Column – maps instance field to column of DB; only need if names are different
        + @JoinColumn – maps foreign key of a managed association
* Hibernate Quety Language – HQL
  + Uses class name instead of table name and property names instead of column names
  + Key words (select, from, where) are not case sensitive but properties such as table and column are
  + HQL FROM: same as select clause 🡪 from <class name> is the same as SELECT \* FROM <table>. Can create aliases 🡪 from <class name> <alias> is from <table name> as <alias>
  + HQL Join: supports inner, left outer, right outer, and full
  + Can use all aggregate functions
  + HQL
  + Query interface
    - Present in org.hibernate package
    - Quesry instance obtained by calling Session.createQuery()
    - Hibernate provides different techniques to query database
      * TypedQuery
      * NamedQuery
      * Criteria API
      * Query query = session.createQuery(); or TypedQuery query = session.createQuery();
    - Running
      * Query.getSingleResult – to get exactly one result
      * Query.getResultList – general use – multiple returns
      * Same for TypedQuery
      * Query.executeUpdate for running delete and update queries
  + HQL Methods
    - List() – return query results as a list but should use getResultList()
    - getResultList() default implementation – execute a SELECT and return results as untyped list
    - General rule
      * If SELECT contains single expression and its an entity, then result is that entity (column names) therefore Select \* From User will return List<User>
      * If SELECT contains single expression and it’s a primitive, then result is that primitive
      * IF SELECT contains multiple expressions then result is Object[] containing the corresponding primitives/entities

**3-14(159)**

* @NamedQuery sets a query to a value to make it more reusable
* @NamedQueries allows you to name multiple

Hibernate Mapping & Relationship Overview

* Hibernate mapping establishes the relationship(association) between tables as attrivbutes in entity classes (model classes) that allows you to easily navigate the assosiactions in models and queries
  + @ManyToOne
  + @OneToMany – required: (cascade = Casecade.TYPE.<type>) If you don’t specify the datatype of the list you have to (targetEntity = <class>.class) => (targetEntity = <Teacher>.class, cascade = CascadeType.ALL)
  + @OneToOne
  + @ManyToMany
* Unidirectional – only one entity (source) has relationship to another entity (target)
* Bidirectional – each entity refers to the other (both are source and target)

**3-15**

* 1-1 can use annotation in either entity, best practice – use in the child (receiving) table
* M-M – must create intermediate JoinTable that holds two foreign keys

H2 DB

* Can be embedded in Java app
* Can be run in client-server mode
* Not a persistent DB; popular in-memory DB
* Featyres
  + Extremely fast, open source, JDBC API
  + Available in embedded and server mode
  + Browser-based console
  + Web console to maintain

JUnit

* Test a single unit of work
  + Basically a method = unit of work
  + Does not depend on completion of other tasks
* Goals
  + Verify that app works
  + Find bugs as early as possible
  + Pieces “work” before putting it together
* Features
  + Before and After methods
  + Easily written tests
  + Graphical output
  + Easy tracking
* Works through annotations
* @BeforeAll – runs before test
* @Test – defines the test to run
* Method – name with “test” in front
  + Assert.asertEquals(<thing 1>, <thing 2>)
  + If not equal it fails
  + assertAll() assertIterableEquals()
  + assertArrayEquals() assertLinesMatch()
  + assertDoesNotThrow() assertNotEquals()
  + assertEquals() assertNotNull()
  + assertFalse() fail()
  + etc
* Annotations
  + @BeforeAll
  + @AfterAll
  + @BeforeEach
  + @AfterEach
  + @Test

**3-16**

* Process for testing for expected data
  + @BeforeAll – Setup
  + @BeforeEach – query data
  + @Test
  + @AfterEach – Reset Query
  + @AfterAll – Close

**3-17**

HTML

* Target attribute can have page open elsewhere
* Title attribute gives you a title when you mouse over
* How to center a div on a page
  + Text-align: center

**3-18**

CSS

Interview shits

* another great application of this is with the .sort() method. you can pass a lambda expression which takes two inputs and returns a positive/negative integer indicating the relative ordering (consistent with .compareTo(), for example).

myList.sort((a,b) -> (b.compareTo(a)));

sorts myList into reverse alphabetical order

This sample creates a dynamic sorting algorithm to sort a list of user-defined objects

public class Driver {

public static void main(String[] args) {

List<Tree> myTrees = new ArrayList<Tree>();

myTrees.add(new Tree("Oak", 10.3));

myTrees.add(new Tree("Maple", 9.9));

myTrees.add(new Tree("Oak", 9.5));

myTrees.add(new Tree("Ash", 5.3));

myTrees.add(new Tree("Maple", 11.2));

myTrees.add(new Tree("Oak", 8.3));

System.out.println("My Trees:");

myTrees.forEach(n -> System.out.println(n));

System.out.println();

// Sorts first by name ascending, then by height descending

System.out.println("Sorting...");

System.out.println();

myTrees.sort((a, b) -> {

int c = a.getName().compareTo(b.getName());

if (c == 0) {

c = Double.compare(b.getHeight(), a.getHeight());

}

return c;

});

System.out.println("My Trees:");

myTrees.forEach(n -> System.out.println(n));

}

}

(edited)

[12:23 PM]

public class Tree {

private String name;

private double height;

public Tree(String name, double height) {

this.name = name;

this.height = height;

}

public String toString() {

return name + "\t" + String.format("%.2f", height);

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getHeight() {

return height;

}

public void setHeight(double height) {

this.height = height;

}

}

Message #✍-notes