03/09/2022 – 03/11/2022

**Concurrency vs. Parallel**

Concurrency: multiple tasks run simultaneously. You can’t tell which task comes first and which task comes after. You can’t tell the order in general.

Parallel: multiple tasks **physically** run simultaneously. This is an implementation level concept. In real time, there are at least two executors.

If there is concurrency, there does not have to have parallel.

If there is parallel, there must have concurrency.

ConcurrentModificationException example:

for (int i : myList) {

myList.remove(i);

}

**Multi-process vs. multi-thread**

Process: an independent execution of instructions with independent memory space, stack, heap, and OS resource. Each process sees a complete memory space (pretend to be the only task of a system). Different processes communicate through inter process communication (explicit IPC).

Thread: an independent execution of instructions with shared memory space. Each thread has its private stack, program counter, and register states. Thread in the same process has shared: heap, static memory segment, OS resource. Communication performed through shared memory read/writes.

Difference: **independent memory space**. If has, process; otherwise, thread.

Multi-process: higher communication overhead, better resource isolation (fault tolerance), higher creation/destroy overhead.

Multi-thread: lower communication overhead, worse resource isolation (fault tolerance), lower creation/destroy overhead.

Multi-thread example:

Public static void main () {

Thread t = new Thread () {

@Override

Public void run () {

System.out.println(“1”);

}

};

t.start();

System.out.println(“2”);

t.join();

System.out.println(“3”);

}

Main thread --- new thread --- print “2” --- join --- print “3” --- exit

| |

---------- print “1” -----

Another example:

Public static void main () {

Thread t = new Thread () {

@Override

Public void run () {

System.out.println(“1”);

}

};

t.start();

System.out.println(“2”);

}

Main thread --- new thread --- print “2” --- exit

| |

---------- print “1” -----

**When will JVM exit? When there are no alive non-daemon threads.**

To make t a daemon thread: t.setDaemon(true);

PS: for JVM, GC is a daemon thread.

**Ways of creating threads and making them run**

1. extends Thread
2. implements Runnable
3. implements Callable

**Methods of thread**

Static methods: sleep(1000), yield() // we can see yield() as sleep(0)

**Synchronization and Race**

Data race: If two “conflicting operations” are in different threads and are not properly synchronized (concurrent), they will introduce data races. In general, two operations conflict with each other if they operate on the same memory location, and at least one of them is a write. Races are mostly treated as bugs in Java programs.

Three factors of data race:

1. more than one operation work on the same memory location
2. at least one operation is a write
3. at least two of those operations are concurrent

**Locks**

**Deadlock**

Condition to form a deadlock:

1. mutual exclusion: at least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.
2. hold and wait or resource holding: a process is currently holding at least one resource and requesting additional resources which are being held by other process.
3. no preemption: a resource can be released only voluntarily by the process holding.
4. circular wait

**Livelock**

**Condition Synchronization**

Producer consumer problem:

* Consumer: if the queue is empty, wait for it to be not empty, then poll one element from it.
* Producer: if the queue is full, wait for it to be not full, then offer one element to it.

03/14/2022

**Volatile keyword**

Ex:

Class SharedObj{

// changes made to sharedVar in one thread

// may not immediately reflect in other thread

static int sharedVar = 6;

}

Suppose two threads are working on the SharedObj and they are running on different processors. Each thread may have its own local copy of shared variables. If one thread modified its value, the changes may not reflect in the original one in the main memory instantly. Now the other thread is not aware of the modified value which leads to data inconsistency.

Class SharedObj{

// volatile keyword makes sure that the changes made in one thread are immediately

// reflect in other thread

static volatile int sharedVar = 6;

}

**volatile vs. synchronized**

1. Mutual exclusion: it means that only one thread or process can execute a block of code (critical section) at a time.
2. Visibility: it means that changes made by one thread to shared data are visible to other threads.

The keyword synchronized guarantees both mutual exclusion and visibility.

The keyword volatile guarantees only the visibility. The value of volatile variables will never be cached and all writes and reads will be done to and from the main memory.

!!! the use of volatile is limited to the cases as most of the times, atomicity is desired.

Ex: a++ is a compound read-modify-write sequence of operations that must execute atomically.

**Sequential consistency**

Writing of a normal variable without any synchronization actions might not be visible to any reading thread.

03/15/2022

**String/StringBuilder/StringBuffer**

* String is immutable
* StringBuilder and StringBuffer are mutable
* StringBuffer is thread safe, StringBuilder is not

Constant pool

* == compares reference address
* Equals() depends on how you implement this method, usually compare the content. Default implementation is same as ==
* new -> in heap
* String a = “aaa”; String b = “aaa”; // in String pool, so a.equals(b) true

03/16/2022

Reviewed:

**Java Collection**

!!! Collection + Map, Map is not a part of Collection

* List – use ArrayList/LinkedList to implement
  + Remove() for both are O(n) not different reason
* Stack and Queue – use Deque to implement
  + For Deque, we use offerFirst(), pollFirst(), offerLast(), pollLast()
  + There are method like pop() but it can be confusing -> prefer not to use
* HashTable vs. HashMap vs. ConcurrentHashMap
* How HashMap works? Ex: find the value of a key
  + Key -> use hashCode() to get hash code
  + Hash code % length (16 buckets) -> get index of the bucket
  + Compare each nodes in that bucket, use equals()
  + If true, return value; if false until the end, does not exist
* Heap – use PriorityQueue
  + We **don’t know** which is larger: left and right

Tomorrow plan:

Comparable and comparator

JVM architecture

Today:

**Java reflection** – Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.

* jave.lang.reflect.\*;
* ex: Class c = Class.ForName("Student”);

Method m[] = c.getDeclaredMethods();

**Singleton design pattern**

* lazy initialization
  + object is created only if it is needed.
  + Implementation: use getInstance() method to return the instance.

public class SingletonExample1 {

private SingletonExample1() {}

private static SingletonExample1 instance = null;

public static SingletonExample1 getInstance() {

if (instance == null) {

instance = new SingletonExample1();

}

return instance;

}

}

* eager initialization
  + object of class is created when it’s loaded to the memory by JVM. It’s done by assigning the reference an instance directly.

public class SingletonExample2 {

private SingletonExample2() {}

private static SingletonExample2 instance = new SingletonExample2();

public static SingletonExample2 getInstance() {

return instance;

}

}

* thread safe
  + synchronized getInstance()

Factory design pattern

Builder design pattern

proxy design pattern

03/17/2022

Reviewed:

**Comparable vs. Comparator**

* Comparable

A comparable object can compare itself with another object.

The class implement Comparable interface to compare its instances.

Override the method compareTo().

class Movie implements Comparable<Movie> {  
 double rating;  
 String name;  
 int year;  
   
 public Movie(String name, double rating, int year) {  
 this.name = name;  
 this.rating = rating;  
 this.year = year;  
 }  
   
 @Override  
 public int compareTo(Movie m) {  
 return this.year - m.year;  
 }  
}

There is only one chance to implement the compareTo() method.

The object itself must know how it is to be ordered, and the sorting of objects needs to be based on natural order.

If a class implements Comparable interface, then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort().

* Comparator

Comparator is external to the element type we are comparing. It’s a separate class.

We can create multiple separate classes to compare by different members.

class RatingCompare implements Comparator<Movie> {  
 @Override  
 public int compare(Movie m1, Movie m2) {  
 if (m1.getRating() < m2.getRating()) return -1;  
 if (m1.getRating() > m2.getRating()) return -1;  
 else return 0;  
 }  
}

We can write more than one custom comparator using different interpretations of what sorting means.

**JVM architecture**

* Class Loader – prepares the Java classes and loads them into main memory
  + Loading
    - Bootstrap Class Loader
    - Extension Class Loader
    - Application Class Loader
  + Linking
    - Verify
    - Prepare
    - Resolve
  + Initialization
    - Initialize
* Runtime Memory – holds the runtime variables and data
  + Method Area
  + Heap Area
  + Stack Area
    - Thread #1
    - Thread #2
    - Thread #N
  + PC Register
    - Thread #1
    - Thread #2
    - Thread #N
  + Native Method Stack
* Execution Engine – executes the Java program
  + Executions Engine
    - Interpreter
    - JIT Compiler
    - Garbage Collector
  + JNI (Native Method Interface)
  + Native Method Library

Tomorrow:

GC

Class Loader in details

Today:

Java Web

**OSI model (7 layers) and TCP/IP model (4 layers)**

|  |  |
| --- | --- |
| 7.Application layer | 1.Application layer |
| 6.Presentation layer |
| 5.Session layer |
| 4.Transport layer | 2.Transport layer |
| 3.Network layer | 3.Internet layer |
| 2.Data Link layer | 4.Network Access layer |
| 1.Physical layer |

**HTTP**

HTTP Request

* HTTP version type
* A URL
* HTTP method
  + GET, PUT, POST…
* HTTP request headers
* HTTP body (optional)

HTTP Response

* HTTP status code
  + 1XX information
  + 2XX success
    - 200 OK, get/put/post
    - 201 successfully created, post
    - 202 successfully received the request, but still processing
    - 204 no content, usually update the resources without changing the current display page, put
  + 3XX redirection
    - 307 temporary redirect
    - 308 permanent redirect
  + 4XX client error
    - 400 bad requests, the server could not understand the request due to invalid syntax
    - 401 unauthorized, the client is not authenticated
    - 403 forbidden, the client has not permission
    - 404 not found, the server cannot find the requested resources
  + 5XX server error
    - 500 internal server error
    - 501 not implemented, method not supported by the server
    - 502 bad gateways
* HTTP response header
* HTTP response body (optional)

03/18/2022 – 03/19/2022

Reviewed:

**Garbage Collection**

Serial GC – single thread

Parallel GC – multi thread

G1 GC – separate all the memory space into different chunks

CMS GC – concurrent mark and sweep GC

PS: Deprecated since java 9, and completely removed in java 14

Young generation / old generation / permanent generation

| | | | |

Eden S0 S1 tenured permanent

**Class Loader in details**

* Loading
* Bootstrap Class Loader
  + Extension Class Loader
  + Application Class Loader
* Linking
  + Verify
  + Prepare
  + Resolve
* Initialization
  + Initialize

Tomorrow:

keywords

03/21/2022

Reviewed:

**Keywords**

* data types
  + byte, short, int, long, float, double, char, boolean (all have a wrapper class)
* flow control
  + if, else, switch, case, default (2 kinds), for, do, while, break, continue, return
  + default
    - new feature in java 8 allows an interface to provide an implementation. We can add default methods in an interface to support lambda expressions.
    - Access modifier when there is no assigned access modifier to a class to its variables. Difference from private, it allows same package access.
* modifier
  + public, private, protected, static, final, //(rest for tomorrow)abstract, synchronized, native, strictfp, transient, volatile
  + final vs. finally vs. finalized

Tomorrow:

Rest of the keywords

Today:

Java application - **Three layers architecture (Controller-Service-Repository)**

Client

|

Web(controller) layer – present the application’s features and data to users.

|

Service(business) layer – business logic implementation, like calculation and evaluation. Also processing the data passing between the other two layers.

|

Data (DAO) layer – interacting with database to save and restore application data.

|

Database

**Inversion of Control** – lifecycle of all objects is controlled by a framework but not developer

**Dependency Injection** – an implantation of IoC, where the control being inverted is setting an object’s dependencies.

* Types: Constructor, Setter, field

Spring 4 main components: @Controller, @Service, @Component, @Repository

Two annotation we gonna use to choose which implementation we want to use

@Primary -> declaration of beans

@Qualifier -> @Autowired

03/22/2022

Reviewed:

**Keywords**

* modifier
  + public, private, protected, static, final, //abstract, synchronized, native, strictfp, transient, volatile
    - synchronized – only one thread can execute at a time.
    - native – non-access modifier that is used to access methods implemented in a language other than Java.
    - strictfp – used for restricting floating-point calculations and ensure the same result on every platform while performing operations in the floating-point variables. Used with classes, interfaces, and methods.
    - Transient – make the variable nor serialized. Serialization is the process of converting an object into a byte stream.
    - Volatile
* exception handling (6)
  + try, catch, finally, throw, throws, assert
    - assert – used for testing. Two ways to use:
      * assert expression;
      * assert expression1 : expression2;
* class related (6)
  + class, package, import, extends, implements, interface
* object related (4)
  + new, instanceof, super, this

Tomorrow:

OOP

Maybe some exceptions

Today:

**Waterfall style** – everything in scheduled, less communication between business and IT

* break a big project into several phases, each phase takes several months
* problem: demand may change over time, chunky

**Agile style**

* Break a project into very small sprints, each sprint takes 1-2 weeks
* Information board – put tickets of this sprint and assign people to work on the tickets. Each ticket has a point (based on difficulty).
* Scrum meeting – usually PM meet with businessperson, decide tickets
* Initial planning meeting – at the beginning of each sprint, to decide value of ticket
* Stand-up meeting – everyday, talk about today & tomorrow work, and any blocks or difficulties

**CI/CD** – continuous integration, continuous delivery/deployment. A method to frequently deliver applications to customers by introducing automation into the stages of app development.

CI – new code changes to an app are regularly built, tested, and merged to a shared repository.

CD (continuous deployment) – automatically release a developer’s changes from the repository to production.

CI/CD tools – Jenkins, GitHub Action

Stages of a CI/CD pipeline

* Source
* Build
* Test
* Deploy

How to build a CI/CD pipeline with GitHub actions? https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/

* Create or choose a repository, and pick a project
* Open GitHub Actions in your repository to start building your CI/CD workflow
* Make changes to your code to trigger your CI/CD pipeline
* Take a look at the workflow visualizer and live logs to get a full look into how your pipeline is running

03/23/2022

Today:

Spring

1. IOC
   1. DI types: Constructor, Setter, field
   2. Bean scope: 5(?)
2. AOP
   1. Definition: aspect-oriented programming

// log - old way  
class Server1 {  
 public void method1(){  
 // log before  
 // business logic  
 // log after  
 }  
  
 public void method2(){  
 // log before  
 // business logic  
 // log after  
 }  
}

Redundant code, hard to change (remove logs)

So, we have AOP:

In AOP, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (cross cutting concerns).

* 1. Key terminology
     1. Aspect
        1. a class that implements enterprise application concerns that cut across multiple classes, such as transaction management, logging, security.
        2. Aspects are implemented regular classes using Spring XML configuration.
        3. Regular classes annotated with @Aspect annotation (@AspectJ style).
     2. Join Point – a point during the execution of a program, such as the execution of a method or the handling of an exception.
     3. PointCut
        1. A predicate that matches join points.
        2. Advice is associated with a pointcut and runs at any join point matches by the pointcut.
        3. Spring framework uses the AspectJ Pointcut expression language.
     4. Advice
        1. Before: advice that executes before a join point
        2. After: advice to be executed regardless of the means by which a join point exits (normal or exception return)
        3. After return: advice to be executed after a join point completes
        4. After throwing: advice to be executed if a method exits by throwing an exception
        5. Around: advice that surrounds a join point such as a method invocation

In three tier layers

Controller -> service -> DAO (repository) -> connect with DB

The proxy class poses as the target bean, intercepting advised method calls and forward the calls to target bean.

* 1. Transaction
     1. How to achieve transaction handling?
        1. Original method to handle transaction:

Connection con = dataSource.getConnection();

Try (connection) {

con.setAutoCommit(false);

// execute sql statement

connection.commit();

} catch (SQLException e) {

connection.rollback();

}

* + - 1. Spring’s Programmatic Transaction Management:

@Service

Public class Userservice {

@Autowired

private TransactionTemplate template;

public Long registerUser(User user) {

Long id = template.execute(status -> {

// execute sql

// insert the user to db and return id

Return id;

})

}

}

* + - 1. Spring’s XML declarative transaction management:

// config in the xml file

<tx: advice id = ‘’, transaction-manager=…>

…

</tx: advice>

<aop>

</aop>

* + - 1. Spring’s @Transactional annotation: (easiest way in three)

Public class UserService {

@Transactional

Public Long registerUser(User user) {

// execute sql

// insert user

Return id;

}

}

* + 1. Propagation level and isolation level

@Service

Public class UserService {

@Autowired

Private InvoiceService service;

@Transactional

Public void invoice() {

service.createpdf();

}

}

@Service

Public class InvoiceService {

@Transactional(propagation = Propagation.REQUIRED)

Public void createPdf() {}

}

-------------------physical only one transaction, but logically there are two-----------------

Get connection,

Start transaction,

Call invoice, create pdf,

Close transaction

------- propagation = Propagation.REQUIRED // if there is already a transaction, use the existed one, otherwise, create a new one--------------

-------default, there is propagation = Propagation.REQUIRED, so you don’t have to explicitly write this--------------

* + - 1. Propagation level
         1. **required**
         2. **required\_new**
         3. support
         4. mandatory
         5. not\_supported
         6. never
         7. nested
      2. Isolation level (in Spring annotation transaction)
         1. default: will use the default isolation level for the RDBMS
         2. read\_uncommitted
         3. read\_committed
         4. repeatable\_read
         5. serializable

Reviewed:

Change plan to Spring setup

Three tier layers

Spring components

@Controller, @Service, @Component, @Repository

Implementation of @Autowired and examples

Tomorrow:

OOP

Maybe some exceptions

03/24/2022

1. Spring MVC
   1. MVC

图

* + 1. Model: contains the data of the application
    2. Controller: business logic of application. @Controller
    3. View: web pages, JSP (old tech)
    4. Front controller: in spring MVC, dispatcher servlet

图

* 1. Flow of Spring MVC
     1. 图
     2. All incoming request is intercepted by the dispatcherServlet that work as the front controller
     3. The dispatcherServlet gets an entry of handler mapping from the xml file and forwards the request to the controller
     4. The controller returns an object of ModelAndView
     5. The dispatcherServlet checks the entry of the view resolver in xml file and invokes the specified view component

1. Spring Boot
   1. Advantage
      1. Provide a flexible way to config the java beans, xml configuration and database transactions
      2. Batch processing and manage rest endpoints
      3. Everything is auto configed
      4. Annotation based spring application
      5. Ease dependency management
      6. Include embedded servlet container -> Tomcat
   2. Spring Boot starters
      1. 图
   3. Auto Config
      1. @EnableAutoConfiguration

public class Application {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

* 1. SpringBootApplication

@SpringBootApplication annotation includes @EnbaleAutoConfiguration, @ComponentScan, @SpringBootConfiguration

图

1. Rest API Design
   1. HTTP method
      1. Get: retrieve

Put: update (whole)

Post: create

Delete: remove

Patch: update (partial)

* + 1. Safe: get
    2. Idempotent: get, put, delete
    3. Cacheable: get, post (in some cases)
    4. Cache-control: (查一下)
  1. HTTP status code
     1. 1XX information
     2. 2XX success

200 OK

201 Ok and created as a result, use with post, and put

202 accepted, but is still processing

204 no content, use with put

* + 1. 3XX redirect

307 temporary redirect

308 permanent redirect

* + 1. 4XX client error

400 bad requests

401 unauthorized

403 forbidden

404 not found

405 method not allowed

* + 1. 5XX server error

500 internal server error

501 request method not implemented

* 1. HTTP URL Design

Handle CRUD (create, read, update, delete) actions using HTTP method:

* + 1. get /api/employees

Retrieves a list. Of employees

get /api/employees/10

retrieve a specific employee

put /api/employees/10

update employee #10

post /api/employees

create a new employee

delete /api/employees/10

delete employee #10

parch /api/employees/10

partial update employee #10

* + 1. each employee has different emails

get /api/employees/10/emails

retrieve a list of emails for employee #10

get /api/employees/10/emails/5

post /api/employees/10/emails

put /api/employees/10/emails/5

patch /api/employees/10/emails/5

delete /api/employees/10/emails/5

* + 1. filtering, sorting, and searching
       1. filtering

get /api/employees?state=open

* + - 1. sorting

get /api/employees?sort=created\_at,salary

// default ascending order, if want descending, use -salary

* + - 1. searching

get /api/employees?q=java

* + - 1. combine all:

get /api/employees?q=java&state=open&sort=salar

* + 1. limit fields

get /api/employees?field=id,department&…

* + 1. auto loading (manager is not a field of employee)

get /api/employees/10?embed=manager.name

get /api/employees/10?expand=manager.name

* 1. Spring RESTful API

@RestController, @Controller, @Service, @Repository,

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @RequestMapping

@PathVariable, @RequestParam,

@RequestBody, @ResponseBody

1. Exception Handling
   1. Exception handling flow

图

@ExceptionHandler

图

@ControllerAdvice

图

@ResponseStatus

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public class NoSuchElementFoundException extends RuntimeException {

// …

}

1. Validation

图

Common validation annotations

@NotNull, @NotEmpty, @NotBlank, @Min, @Max, @Pattern, @Email …

图

@Valid, @Validated

1. RestDemo

Reviewed:

**OOP**

Abstraction – hide all the internal implementation

* Abstract class – contains abstract methods
* Interface – only abstract methods and default methods

Encapsulation – hide the data in a single entity or unit along with a method to protect information from outside

* Declare all the variables be private
* Provide getter and setter

Inheritance

* Extends – for class, can only extends one class
* Implements – for interface, can implement multiple interfaces

Polymorphism – the ability of an object to take many forms

* Override – implement inherited methods in subclasses (both abstract and non-abstract methods can be overridden, must implement abstract methods in subclasses)
* Overloading – methods have same name but different signatures (number of parameters or type of parameters)

Tomorrow:

Exception and its examples

Weekend – notes for spring, and spring demos

03/25/2022

1. Swagger – Documentation framework

Swagger is an open-source project used to generate the REST API documents for RESTful services.

图

1. Review
   1. Spring evolution

Spring core (applicationContext.xml) -> Spring MVC (dispatcher servlet) -> spring boot (embedded web container -> Tomcat) -> spring cloud -> Spring Cloud Data Flow

Spring Security, Spring Data, Spring Batch, webflux…

* 1. IOC
     1. Factory design, dependency injection…
     2. 图
  2. DI
     1. Constructor
     2. Filed
     3. Setter
  3. Bean scope
     1. Single (default)
     2. Prototype
     3. Request
     4. Session
     5. Global session
  4. @Autowired
     1. By type
     2. @Autowired(required = false), the bean can be null
     3. @Qualifier(value = “myBean”)
  5. @Resource
     1. Find by name first, then by type
     2. @Nullable
  6. @Component
     1. Bean managed by container
     2. @Controller, @Service, @Repository, @RestController
     3. @ResponseBody + @Controller = @RestController
  7. @Scope(“singleton/prototype”)
  8. @Configuration
     1. @Import(MyConfig2.class) -> include tag in C
  9. @ComponentScan(“package”)
  10. @Bean
  11. MVC
      1. Model view controller
      2. Dispatcher servlet
  12. Boot
      1. Auto config
      2. Starter
      3. Tomcat
      4. Annotation based…
  13. RESTful API annotation

@RestController, @Controller, @Service, @Repository,

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @RequestMapping

@PathVariable, @RequestParam,

@RequestBody, @ResponseBody

* 1. Lombok (not in RestDemo)
     1. @Data
     2. @AllArgsConstructor
     3. @NoArgsConstructor

// code here

* 1. Exception
     1. @Exceptionhandler
     2. @ControllerAdvice
     3. @ResponseStatus
  2. Validation

@ NotNull, @NotEmpty, @NotBlank, @Min, @Max, @Pattern, @Email…

* 1. @Value(“”) // inject value to a specific field

int mailAddress;

* 1. @ConfigurationProperties(prefix = “user”)
  2. @Profile(“dev”)

application-test1.properties

application-test2.properties

application-product.properties

application-qa.properties

application-dev.properties

Reviewed:

**Spring related**

I reviewed my Spring-related knowledge while completing my homework.

@Controller, @Service, @Repository, @RestController

Model-View-Controller

RESTful API annotations

Model interface – addAttributes

Lombok – auto generating setter and getter

Not Lombok but kind of similar(?) @ToString

Validation – @NotNull, @Min, @Max

**Exceptions**

Checked vs. unchecked

How to handle – try catch, throws

Tomorrow:

Rest of Exceptions

03/28/2022

Reviewed:

**Multiple exceptions**

* try {} catch (IOException ioe) {} catch (SQLException sqle) {} catch …
* try {} catch (IOException | SQL Exception | …) {}
* try { Connection con = DataDriver.getConnection();} catch (IOException ioe) {} catch (Exception ex) {} finally { if (con != null) {con.close();}}// from child to parent

**Generics**

* generic method

public <T> List<T> fromArrayToList(T[] a) {

return Arrays.stream(a).collect(Collectors.toList());

}

// <T> signature implies the method will be dealing with generic type T. it is needed even if return void.

* + Upper-bounded type <T extends Number> - restrict the types that a method accepts
  + Multiple bounds <T extends Number & Comparable>

If one of the types that are extended by T is a class, we have to put it first in the list of bounds.

* Wildcards ? – refers to an unknown type

public static void paintAllBuildings(List<? extends Building> buildings) {}

// the method can work with type Building and all its subtypes.

Tomorrow:

Probably more on generics and play with some examples

Iostream

03/29/2022

Today:

Security

1. Authentication vs. authorization
   1. Authentication – The process of checking validity of a user or request. To verify who you are.
      1. Username + password
   2. Authorization – To give someone the permission to do some actions. After authentication, you automatically have authorization.
      1. A role, ex: admin user
      2. A token
2. Security
   1. Data at rest
   2. Date in transit
3. Encryption
   1. Symmetric – 123 -> key -> &\*%^&% -> key -> 123
      1. For data
   2. Asymmetric – 123 -> key1 -> &\*%^&% -> key2 -> 123
      1. For signature, authentication
4. Hashing – used to verify
   1. Same text, same hex code
   2. Ex: used to save user password, and hash the user input each time to compare
   3. 123 -> md5/SHA (algorithm) -> 128 bits (hex)
5. Encoding
   1. URL encoding
      1. Convert ex the Chinese character into a format that can be transmitted over the internet
      2. 123a-z中文 -> 123a-z
   2. File encoding
      1. Binary -> characters
      2. Ex: Base64
6. SSH (user -> Linux)
   1. Server (has public key of a user) <- SSH (private)
      1. Server encrypt with public key “hello”
      2. SSH decrypt with private key
      3. Server verify your role
7. API (?) – we protect the server

GET /api/user

GET /api/user?pageNo=100

POST /api/user {username, password, tel…}

PUT

DELETE

POST /api/login {username;passname} -> token

* 1. Validate input controller – ex: SQL injection (drop user, truncate admin, or 1==1 -- ; , () potential special characters to hack), XSS attack
  2. HTTP + Security (TLS > SSL)
     1. TLS handshakes
  3. Token -> JWT (Jason Web Token)

1. Oauth2
   1. Flow
2. CORS (cross origin resource sharing) \*\* RESTful + SPA(angular react)
3. SSO (Single Sign On) -> login once, login everything
4. LDAP, Active Directory (AD) -> Directory DB.

Reviewed:

**Generics**

Restrict the types that a method accepts

* Upper bound – a method accepts a type and all its subclasses
  + <T extends Number>
  + <T extends Number & Comparable> // if one of the types that are extended by T is a class, we have to put it first in the list of bounds
* Lower bound – a method accepts a type and all its superclasses
  + <? super T>
* Wildcards – are represented by the question mark “?” in java
  + Note: Object is the supertype of all Java classes, but a collection of Object is not the supertype of any collection. Ex: List<Object> is not the supertype of List<String>.
  + Same rule applies to any collection of a type and its subtypes.
  + Ex:

public static void paintAllBuildings(List<? extends Building> buildings) {}

// the method can work with type Building and all its subtypes.

* Genetics and Primitive Data Types
  + Generics are compile-time feature: the type parameter is erased and all generic types are implemented as type Object
  + The type parameter cannot be a primitive type

**IO Stream**

Standard input stream: System.in

Standard output stream: System.out

Standard error stream: System.err

* Types of Streams
  + Depending on the type of operations
    - Input Stream – used to read data that must be taken as an input from a source array or file or any peripheral device.
      * Eg., FileInputStream, BufferedInputStream, ByteArrayInputStream.
    - Output Stream – used to write data as outputs into an array or file or any output peripheral device.
      * Eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream.
  + Depending on the types of file
    - ByteStream – used to process data byte by byte (8 bits).
      * Eg., FileInputStream, FileOutputStream
    - CharacterStream – automatically allows us to read/write data character by character. Characters are stored using Unicode conventions.
      * Eg., FileReader, FileWriter

03/30/2022

Reviewed:

Java 8 features

* Lambda – (arguments) -> {body}
  + How to write: just like lambda function in python
  + Functional programming
  + Less code
* Functional interface
  + Predicate – public boolean test(T t)
  + Function – public R apply(T t)
  + Consumer – public void accept(T t)
  + Supplier – public R get(T t)
* Optional
  + used to prevent NPE (NullPointException)
  + If (obj == null) {} else {} // don’t have to use these
* Stream API
  + Intermediate operation: return a stream as result
    - Map, flatmap, filter…
  + Terminal operation: return nun-stream
    - forEach, collect

Tomorrow:

SQL related

03/31/2022

Reviewed:

Database Management System

File system vs. DBMS

|  |  |
| --- | --- |
| File System | DBMS |
| Manage and organize the files in storage medium | Manage the database |
| Redundant data | No redundant data |
| No efficient query processing | Efficient query processing |
| Less data consistency | More data consistency |
| Less security | More security |
| Less expensive | Higher cost |

Database Normalization

* 1NF – each table cell should contain a single value and each record need to be unique
* 2NF – 1NF + single column primary key
* 3NF – 2NF + has no transitive functional dependencies

SQL vs. no SQL

|  |  |
| --- | --- |
| SQL | No-SQL |
| Relational database | Non-relational database |
| Pre-defined schema | Dynamic schema |
| Vertical scaling | Horizontal scaling |
| ACID | CAP |
| Not suited for hierarchical data store | Suited for hierarchical data store |

Tomorrow:

Details in SQL and no SQL

04/01/2022

Reviewed:

**No SQL**

Major categories of non-relational database

1. Document based
2. Key-value based
3. Columnar
4. Graph based

CAP – consistency, availability, and partition tolerance

* Consistency: all clients always have the same view of the data
* Availability: each client can always read and write
* Partition tolerance: the system works despite physical network partitions

ACID – atomicity, consistency, isolation, and durability

Sharding: distribute a single logical database across a cluster of machines

Replica: copies of the original database

Concurrency

* Dirty data: read uncommitted data from another transaction
* Non-repeatable read: read committed data from an update query from another transaction
* Phantom read: read committed data from an insert or delete query from another transaction
* Isolation level

Tomorrow:

AWS

04/04/2022

Reviewed:

**AWS**

EC2: a service that allows for using virtual machines. It provides user servers with CPU, memory, disk, network, and OS.

S3 (Simple Storage Service): file storage using bucket and key to indicate the id. Fast and lots of space.

SQS (Simple Queue Service): message Queue. Very simple one.

SNS (Simple Notification Service): notification service, can send SMS, Email, SQS, HTTP etc.

Lambda: a computing platform that allows you to run a price of code written on one of the supported programming languages. It hosts functions which could be triggered by other services. It is serverless. Could be used for small tasks which don’t need a dedicated server.

RDS (Relational Database Service): a collection of managed services that makes it simple to set up, operate, and scale database in the cloud. RDBMS hosts, like Oracle, MySql, PostgreSQL, etc.

ECS (Elastic Container Service): Orchestration and management of Docker containers.

ECR (Elastic Container Registry): Registry of Docker Images.

04/05/2022

Reviewed:

**Web Services**

REST: an uniformed, resource-based API which uses HTTP to identify the operations and resources.

HTTP methods/verbs: GET, POST, PUT, DELETE, PATCH, OPTIONS

CRUD (create, read, update, delete)

Safe: get

Idempotent: get, put, delete

Cacheable: get, post (in some cases)

HTTP status code

* 1XX information
* 2XX success
  + 200 OK
  + 201 Ok and created as a result, use with post, and put
  + 202 accepted, but is still processing
  + 204 no content, use with put
* 3XX redirect
  + 307 temporary redirect
  + 308 permanent redirect
* 4XX client error
  + 400 bad requests
  + 401 unauthorized
  + 403 forbidden
  + 404 not found
  + 405 method not allowed
* 5XX server error
  + 500 internal server error
  + 501 request method not implemented

04/06/2022

Reviewed:

**Thread**

Thread states

New – thread create, not yet start

Runnable – executing in JVM

Blocked – wait for a monitor lock to enter synchronized block or method

Waiting – object.wait with no timeout, thread.join() with no timeout, park()

Timed waiting – thread sleep, Object.wait() with timeout, thread with timeout, park

Terminated – thread has completed

Runnable interface: only declare one method void run(), no return, no exception.

Callable interface: only declare one method T call() throws Exceptions().

Tomorrow:

Thread pool

04/07/2022

Reviewed:

ThreadPoolExecutor – a utility class which can create FixedSizeThreadPool, CachedThreadPool, ScheduledThreadPool by using ThreadPollExecutor class internally.

* corePoolSize
* maximumPoolSize
* keepAliveTime
* timeUnit
* workQueue
* threadFactory
* handler
  + abortPolicy
  + callerRunPolicy
  + discardPolicy
  + discardOldestPolicy

04/08/2022

Reviewed:

**Security**

Authentication: checking validity of a user or request

Authorization: giving someone the permission to do something

Encryption

Hashing: used to verify. Same text will have same hex code.

Encoding

* url encoding, convert special characters into a format that can be transmitted over the internet
* file encoding, ex base64

HTTP+Security: HTTP over TLS/SSL

* TLS (transport layer security)
* TLS handshake: two communicating sides exchange messages to acknowledge each other, verify each other, establish the encryption algorithms they will use, and agree on session keys.
* SSL (secure sockets layer): original encryption protocol developed for HTTP.

04/11/2022

Reviewed:

**Spring**

Spring MVC

* Follows Model-View-Controller design pattern
* DispatcherServlet: a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.
* Advantages
  + It separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
  + Facilitates fast and parallel development

Spring IoC

* IoC (Inversion of Control): a principle which transfers the control of objects or portions of a program to a container or framework.
* DI (Dependency Injection): connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler righter than by the objects themselves.
* ApplicationContext interface represents the IoC container. It is responsible for instantiating, configuring, and assembling objects knows as beans.

Spring ORM

* Spring-ORM is an umbrella module that covers many persistence technologies, namely JPA, JDO, Hibernate and iBatis.

Spring AOP

* AOP (Aspect Oriented Programming): a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does this by adding additional behavior to existing code without modifying the code itself.

Group meeting:

Education – 周三会讲

某技术在哪儿用的？怎么用的？为什么是用这个技术？例子

电话号码会邮件给

Location和时间？问一下中间空几个月不在美国

Challenge：performance？

问问题 team size/tech, working environment

Location和时间？寒暑假，问一下中间空几个月不在美国

第一个项目想不出function

04/12/2022

Reviewed:

Spring AOP

* Aspect: a module which has a set of APIs providing cross-cutting requirements.
* Join Point: a point in your application where you can plug-in the AOP aspect.
* Advice: the actual action to be taken either before or after the method execution.
  + Before, after, after-returning, after-throwing, around.
* Pointcut: a set of one or more join point where an advice should be executed.

Spring Boot

* An open source, microservice-based Java web framework.
* Benefits
  + Standalone application: can simply build the application jar and run the application with no need to customize the deployment.
  + Embedded servers: comes with prebuilt Tomcat, Jetty and Undertow application servers that do not require further installation to use.
  + Auto configurable: Spring and other 3rd party frameworks will be configured automatically.
* @SpringBootApplication
* @EnableAutoConfiguration
* @ComponentScan

Parallel Stream – Java 8 new feature

* Sequential Streams: by default, any stream operation in Java is processed sequentially.
  + listOfNumbers.stream().forEach(number -> System.out.println(number+” ”+Thread.currentThread().getName()));
  + print in order
* Parallel Streams: any stream can be transformed from sequential to parallel.
  + listOfNumbers.parallelStream().forEach(number -> System.out.println(number+” ”+Thread.currentThread().getName()));
  + parallel streams enable us to execute code in parallel on separate cores. The result is the combination of each individual outcome.
  + The order of execution of out of our control. It may change every time we run the program.

Stream.filter() – Java 8 new feature

* Allows us to filter elements of a stream that match a given Predicate.
  + someNumbers.stream().filter(c -> c.getPoints() > 100).collect(Collectors.toList());

tomorrow:

Hibernate

Design patterns

04/13/2022

**Hibernate**

* JDBC (Java Database Connectivity): provides a set of APIs for accessing the relational databases from java program.
* JPA (Java Persistence API): a specification that defines how to persist data in Java application. Hibernate is a standard implementation of the JPA specification.
* ORM (Object Relational Mapping): a programming technique for converting data between relational databases and object-oriented programming languages.
  + Advantages
    - Let business code access objects rather than DB tables.
    - Hides details of SQL queries from OO logic.
    - No need to deal with the database implementation.
    - Transaction management and automatic key generation.
* Hibernate: an ORM solution for Java. It maps Java classes to database tables and from Java data types to SQL data types and retrieves the developer from 95% of common data persistence related programming tasks.
  + Advantages
    - Takes care of mapping Java classes to database tables using XML files and without writing any line of code
    - Provides simple APIs for storing and retrieving Java objects directly to and from the database
    - Does not require an application server to operate
  + Uses existing Java APIs like JDBC, JTA (Java Transaction API), and JNDI (Java Naming and Directory Interface)

**Design Patterns**

**Singleton**

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.

* Eager instantiation
* public class Singleton {  
   private static Singleton *instance* = new Singleton();  
    
   private Singleton() {  
   System.*out*.println("Singleton being initialized.");  
   }  
    
   public Singleton getInstance(){  
   return *instance*;  
   }  
  }
* Lazy instantiation
* public class Singleton {  
   private static Singleton *instance* = null;  
    
   private Singleton() {  
   System.*out*.println("Singleton being initialized.");  
   }  
    
   public Singleton getInstance(){  
   if (*instance* == null) {  
   *instance* = new Singleton();  
   }  
   return *instance*;  
   }  
  }
* thread-safe
* public class Singleton {  
   private static Singleton *instance* = null;  
    
   private Singleton() {  
   System.*out*.println("Singleton being initialized.");  
   }  
    
   public Singleton getInstance(){  
   Singleton localRef = *instance*;  
   if (localRef == null) { // check that variable is initialized without obtaining the lock  
   synchronized (this) { // obtain the lock  
   localRef = *instance*;  
   if (localRef == null) { // double-check whether the variable has already been initialized  
   *instance* = localRef = new Singleton();  
   }  
   }  
   }  
   return localRef;  
   }  
  }

**Factory**

We create object without exposing the creation logic to the client and refer to newly created object using a common interface.

public interface Shape {  
 void draw();  
}

public class Rectangle implements Shape {  
 @Override  
 public void draw() {  
 System.*out*.println("Rectangle!");  
 }  
}

public class Square implements Shape {  
 @Override  
 public void draw() {  
 System.*out*.println("Square!");  
 }  
}

public class Circle implements Shape {  
 @Override  
 public void draw() {  
 System.*out*.println("Circle!");  
 }  
}

public class ShapeFactory {  
 public Shape getShape(String shapeType) {  
 if(shapeType == null){  
 return null;  
 }  
 if(shapeType.equalsIgnoreCase("CIRCLE")){  
 return new Circle();  
  
 } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
 return new Rectangle();  
  
 } else if(shapeType.equalsIgnoreCase("SQUARE")){  
 return new Square();  
 }  
   
 return null;  
 }  
}

04/17/2022

Reviewed:

**Builder**

* Method chaining: used to invoke multiple methods on the same object which occurs as a single statement. It is implemented by a series of methods that return the “this” reference for a class instance.
* final class Student {  
   private final int id;  
   private final String name;  
   private final String address;  
    
   public Student(Builder builder) {  
   this.id = builder.id;  
   this.name = builder.name;  
   this.address = builder.address;  
   }  
     
   public static class Builder {  
   private int id;  
   private String name;  
   private String address;  
     
   public static Builder newInstance() {  
   return new Builder();  
   }  
     
   private Builder() {}  
     
   private Builder setId(int id) {  
   this.id = id;  
   return this;  
   }  
    
   private Builder setName(String name) {  
   this.name = name;  
   return this;  
   }  
    
   private Builder setAddress(String address) {  
   this.address = address;  
   return this;  
   }  
     
   public Student build() {  
   return new Student(this);  
   }  
   }  
  }

class StudentReceiver {  
 private volatile Student student;  
  
 public StudentReceiver() {  
 Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 student = Student.Builder.*newInstance*()  
 .setId(1)  
 .setName("K")  
 .setAddress("123 st")  
 .build();  
 }  
 });  
  
 Thread t2 = new Thread(new Runnable() {  
 @Override  
 public void run()  
 {  
 student = Student.Builder.*newInstance*()  
 .setId(2)  
 .setName("Shyam")  
 .setAddress("Delhi")  
 .build();  
 }  
 });  
  
 t1.start();  
 t2.start();  
 }  
  
 public Student getStudent()  
 {  
 return student;  
 }  
}  
  
class BuilderDemo {  
 public static void main(String args[])  
 {  
 StudentReceiver sr = new StudentReceiver();  
 System.*out*.println(sr.getStudent());  
 }  
}