Midterm Prep

Tuesday, February 18, 2020 7:04 PM

Week 1 :

- Lecture slides
  - Key Concepts
    - Measure of Good Software
      - Modularity
        - Encapsulation
          - Declare the variables of a class as private.
          - Provide public setter and getter methods to modify and view the variables values.
        - Abstract Data Type
      - Cohesion : how focused are the responsibilities of a module
      - Coupling : dependency between modules
      - Modifiability and Testability
      - Safety
    - Access levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

  - Classes, Objects, Fields, Methods
  - Constructors
    - no return
    - constructor chaining : calls another constructor of same class
    - If you do not implement any constructor in your class, Java compiler inserts a default constructor

- this.x = x
  - because it will go outside the scope of constructor - not local
- super()
  - Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class.
  - 1) super()(or parameterized super must be the first statement in constructor otherwise you will get the compilation error: “Constructor call must be the first statement in a constructor”
  - 2) When we explicitly placed super in the constructor, the java compiler didn’t call the default no-arg constructor of parent class.
  - https://beginnersbook.com/2014/07/super-keyword-in-java-with-example/
  - method overriding
    - What if the child class is not overriding any method: No need of super
    - Access modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class.
    - The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class).The data types of the arguments and their sequence should exactly match.
    - private, static and final methods cannot be overridden as they are local to the class.
      - this.getTickets().add(newTicket);
      - was getting error when doing super.add(newTicket) because it was private or something

        - we use getters and setters
      - Binding of overridden methods happen at runtime which is known as dynamic binding
      - Overriding method (method of child class) can throw **unchecked exceptions**, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw **checked exceptions** that are new or broader than the ones declared by the overridden method.
      - If a class is extending an **abstract class** or implementing an **interface** then it has to override all the abstract methods unless the class itself is a abstract class.
  - this() and super() should be the first statement in the constructor code. If you don't mention them, compiler does it for you accordingly.
    - https://beginnersbook.com/2013/03/constructors-in-java/
    - Default constructor
  - constructor overloading : more than one constructor with different parameters list
  - static constructor - not in java
- Working with related classes
- Collections
  - LinkedList
- this
  - https://javabeginnerstutorial.com/core-java-tutorial/this-keyword-java/
    - field
    - constructor
    - method
    - as a method parameter

        - as a method parameter
    - Interfaces
        - **Interface can only have abstract methods, they cannot have concrete methods**
        - **the keyword ‘abstract’ is optional to declare a method as an abstract because all the methods are abstract by default**
            - `void display();`
        - **they have public members**
        - **implementing classes inherit the type**
    - Abstract classes
        - **Abstract class can be extended(inherited) by a class or an abstract class**
        - **In abstract class, the keyword ‘abstract’ is mandatory to declare a method as an abstract**
        - **Abstract class can have both abstract and concrete methods**
        - can abstract implement interface
            - **abstract** class **can implement** an **interface**, and not provide implementations of all of the **interface's** methods. It is the responsibility of the first concrete class that has that **abstract** class as an ancestor to **implement** all of the methods in the **interface**.
    - -
    - Others
        - -
        - Encapsulation
            - binding object state(fields) and behaviour(methods) together.
            - encapsulation is known as **data hiding.**
            - **public getter and setter methods to update and read the private data fiels.**
        - When do you use Java's @Override annotation and why?
            - compiler checking to make sure you actually are overriding a method when you think you are - spelling mistakes, matching parameters

matching parameters
    - code easier to understand
- method overloading
    - https://beginnersbook.com/2013/05/method-overloading/
    - declaring same method with different parameters - 3 ways
        - number of parameters
        - data type of parameters
        - sequence of data type of parameters
    - if two methods have same name, same parameters and have different or same return type, then this is not a valid method overloading
    - Method overloading is an example of static polymorphism
    - static polymorphism - also known as compile time binding or early binding
    - static binding happens at compile time, eg. method overloading.
    - type promotion - type of smaller size to type of bigger size
        - byte → short → int → long
          short → int → long
          int → long → float → double
          float → double
          long → float → double
        - boolean 1 - byte 8 - char 16 - short 16 - int 32 - long 64 - float 32 - double 64
            - int : 32
              bit - *-2,147,483,648* to *2,147,483,647* (-2^31 to 2^31 – 1).
- variable num1 = (expression) ? value if true : value if false
- Final
    - https://www.geeksforgeeks.org/final-keyword-java/
    - Final variable - to create constant variable

    - Final Methods - prevent method overriding
    - Final classes - prevent inheritance
- toString()
- Static fields and methods ; static block
    - static variables also called Class variables ; shared among all instances of class
    - can access it without object.
    - https://beginnersbook.com/2013/04/java-static-class-block-methods-variables/
- for
    - for (i = 0, j = 0; i < lname.size() || j < rname.size(); i++, j++)
    - ```
      int arr[]={2,11,45,9};
       for (int num : arr) {
        System.out.println(num);
       }
      ```
    - -
- ```
  public class Employee {
       private Assignments[] assignments;
  }
  public class Project {
       private String name;
       private double budget;
       private Assignment[] assignments;
  }
  public class Assignment {
    private Employee employee;
    private Project project;
    private double load;
    // constructors and getters and setters
  }
  ```
- Arrays
    - int intArray[];
      or int[] intArray;
    - Object[] ao,   // array of Object
      Collection[] ca;
    - int[] intArray = new int[20];
    - int[] intArray = new int[]

- int[] intArray = new int[]
  { 1,2,3,4,5,6,7,8,9,10 };
- int[][] intArray = new int[10][20]; //a 2D array or matrix
- ```
  int arr[][] =
  { {2,7,9},{3,6,1},{7,4,2} };
  ```
  - ```
    for (int i=0; i< 3 ; i++) {
        for (int j=0; j < 3 ; j++)
         System.out.print(arr[i][j] + "
    ");}}
    ```
  - 2 7 9
    3 6 1
    7 4 2
- ```
  int[][][] arr = { { { 1, 2 }, { 3,
  4 } }, { { 5, 6 }, { 7, 8 } } };
  ```
  - arr[0][0][0] = 1
    arr[0][0][1] = 2
    arr[0][1][0] = 3
- The elements in the array allocated by *new* will automatically be initialized to **zero** (for numeric types), **false** (for boolean), or **null** (for reference types)
- When you clone a single dimensional array, such as Object[], a “deep copy” is performed with the new array containing copies of the original array’s elements as opposed to references.
- A clone of a multidimensional array (like Object[][]) is a “shallow copy” however, which is to say that it creates only a single new array with each element array a reference to an original element array but subarrays are shared.
- Two arrays are said to be equal if both of them contain same set of elements, arrangements (or permutation) of elements may be different though.

- Vector vs ArrayList in Java
  ```

- Vector vs ArrayList in Java
  - Vector is **synchronized**, which means only one thread at a time can access the code, while arrayList is **not synchronized**, which means multiple threads can work on arrayList at the same time.
  - Vector can use both **Enumeration and Iterator** for traversing over elements of vector while ArrayList can only use **Iterator** for traversing.
  - ArrayList<T> al = new ArrayList<T>();
    Vector<T> v = new Vector<T>();
  - ArrayList<Integer> arrli = new ArrayList<Integer>();
- Collection
  - basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
  - https://www.geeksforgeeks.org/collections-in-java-2/
  - **Collection** : Root interface with basic methods like add(), remove(), contains(), isEmpty(), addAll(), ... etc.
  - **Set** : Doesn't allow duplicates. Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based). Note that TreeSet implements **SortedSet**.
  - **List** : Can contain duplicates and elements are ordered. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)
    - public abstract interface List extends Collection

extends Collection

- List a = new ArrayList();
  List b = new LinkedList();
  List c = new Vector();
  List d = new Stack();
- List<Obj> list = new ArrayList<Obj>();

- **Queue** : Typically order elements in FIFO order except exceptions like PriorityQueue.
- **Deque** : Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.
- **Map** : Contains Key value pairs. Doesn't allow duplicates. Example implementation are HashMap and TreeMap.
  TreeMap implements **SortedMap**.
- The difference between Set and Map interface is that in Set we have only keys, whereas in Map, we have key, value pairs.

Part 2

- Packages
  - The wild card import like package.* should be used carefully when working with subpackages. For example: Lets say: we have a package **abc** and inside that package we have another package **foo**, now **foo** is a subpackage.
    - classes inside abc are: Example1, Example 2, Example 3
    - classes inside foo are: Demo1, Demo2
    - import abc.*;
      - only import classes Example1, Example2

        - only import classes Example1, Example2 and Example3 but it will not import the classes of sub package.
    - import abc.foo.*;
        - will import Demo1 and Demo2 but it will not import the Example1, Example2 and Example3.
- Garbage Collection
    - The process of removing unused objects from heap memory is known as **Garbage collection** and this is a part of memory management in Java.
    - when does java perform garbage collection ?
        - when the object is no longer reachable
            - obj = null
        - when one reference is copied to another reference
            - obj2 = obj1
    - How to request JVM for garbage collection ?
        - System.gc();
        - finalize() : invoked just before an object is destroyed by java garbage collection process
- Inner class
    - 4 ways to define inner classes
        - Inner class
            - access to outer class members including private members too.
            - To instantiate an instance of inner class, there should be a live instance of outer class.
            - **Instantiating an inner class from outside the outer class Instance Code**
        - Method - local inner class
            - The inner class can use the local variables of the method (in which it is present), only if they are marked final.
        - Anonymous inner class
        - static nested class
- Serialization

- Serialization is a mechanism to convert an object into stream of bytes so that it can be written into a file, transported through a network or stored into database. De-serialization is just a vice versa.

- Generics
  - ```
    class Test<T, U> {
    T obj1;  // An object of type T
    U obj2;  // An object of type U
    ```
  - ```
    class name<T1, T2, ..., Tn> { /* ... */ }
    ```
  - ```
    BaseType <Type> obj = new BaseType <Type>()
    ```
    - ```
      Test <String, Integer> obj = new
      Test<String, Integer>("GfG", 15);
      ```
  - ```
    // A Generic method example
    ```
    - ```
      static <T> void genericDisplay (T element)
      {…}
      ```
    - ```
      public static <T extends Comparable<T>> int
      compare(T t1, T t2){
      ```
    - ```
      public static double sum(List<Number> list){
      ```
      - ```
        for(Number n : list){ ….
        ```
    - ```
      public static < E > void printArray( E[]
      inputArray ) {
            // Display array elements
            for(E element : inputArray) {
      ```
  - Interface
    - ```
      interface MyList<E,T> extends List<E>{
      ```
  - most commonly used type parameter names are:
    - E - Element (used extensively by the Java Collections Framework)
    - K - Key
    - N - Number
    - T - Type
    - V - Value
    - S,U,V etc. - 2nd, 3rd, 4th types
  - List<?>
  - **List<? extends Number>**
  - ```
    public static void addNumbers(List<? super Integ
    er> list) {
    ```

Part 3

- Iterators
  - Iterators are used in Collection framework in Java to retrieve elements one by one. There are three iterators.
    - Enumerators
    - Iterator
      - Iterator object can be created by calling *iterator()* method present in Collection interface
        - Iterator itr = c.**iterator**();
      - Iterator interface defines **three** methods:
        - **public boolean hasNext();**
        - **public Object next();**
        - **public void remove();**
      - **Limitations of Iterators**
        - Only forward direction iterating is possible.
        - Replacement and addition of new element is not supported by Iterator.
    - ListIterator
      - ```
        // Java program to demonstrate ListIterator
        import java.util.ArrayList;
        import java.util.ListIterator;
        ```
  - We don’t create objects of Enumeration, Iterator, ListIterator because they are interfaces. We use methods like elements(), iterator(), listIterator() to create objects. These methods have anonymous Inner classes that extends respective interfaces and return this class object.
- Exception Handling
  - types of exceptions
    - checked exceptions
      - all exceptions other than runtime
        ```

        - exceptions. If not handled - compilation error.
    - unchecked exceptions
        - runtime exceptions
- try-catch-finally
    - A **finally block** contains all the crucial statements that must be executed whether exception occurs or not.
        - The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.
            - In this case **finally block runs**. Control first jumps to finally and then it returned back to **return statement.**
    - finally and close()
    - if an exception occurs then the rest of the try block doesn't execute and control passes to catch block.
- throw exception
    - `throw new exception_class("error message");`
- throws clause
    - **Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.
    - forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.