USN | | | | | | | | | | |

Internal Assessment Test 2 – June 2022

Sub:	Advanced Java and J2EE					Sub Code:	18CS644	Branch:	CSE / ISE
Date:	09.06.2022	Duration:	90 mins	Max Marks:	50	Sem / Sec:	6 A,B,C		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Explain Map and Map.Entry interface ***map*** is an object that stores associations between keys and values, or *key/value pairs* Map.Entry Describes an element in a map. This is an inner class of **Map**. **V get(Object** key) : obtains value by passing the key **V put(K** key, **V** value) : puts value in map by specifying key and value. boolean containsKey(Object key) boolean containsValue(Object value) **Set**<**K**> **keySet**() **Set**<**Map.Entry**<**K**,**V**>> **entrySet**() **Collection**<**V**> **values**() - Obtains collection view of Keys and elements or just valuesin Map. The Map.entrySet method returns a collection-view of the map, whose elements are of this class From an iterator over Map.entrySet, reference to Map.Entry can be obtained. boolean **equals(Object** o) Compares the specified object with this entry for equality. **K getKey**() Returns the key corresponding to this entry. **V getValue**() Returns the value corresponding to this entry. Int **hashCode**() Returns the hash code value for this map entry. **V setValue(V** value) Replaces the value corresponding to this entry with the specified value class HashMap<K, V> HashMap Extends **AbstractMap to use a hash table.** allows the execution time of **get( ) and put( ) to remain** constant even for large sets **HashMap**():Constructs an empty HashMap Default : capacity (16), load factor (0.75). **HashMap**(int initialCapacity) **HashMap**(int initialCapacity, float loadFactor) **HashMap(Map**<? extends **K**,? extends **V**> m) import java.util.*;	[05]	CO2	L2

	public class hashmapdemo { public hashmapdemo(){ **// Create a hash map.** HashMap<String, Double> hm = new HashMap<String, Double>(); **// Put elements to the map** hm.put("John Doe", new Double(3434.34)); hm.put("Tom Smith", new Double(123.22)); hm.put("Jane Baker", new Double(1378.00)); hm.put("Tod Hall", new Double(99.22)); hm.put("Ralph Smith", new Double(-19.08)); Set<Map.Entry<String, Double>> set = hm.entrySet(); **// Get a set of the entries.** for(Map.Entry<String, Double> me : set) {**// Display the set.** System.out.print(me.getKey() + ": "); System.out.println(me.getValue()); } System.out.println(); // Deposit 1000 into John Doe's account. double balance = hm.get("John Doe"); hm.put("John Doe", balance + 1000); } Output: Tod Hall: 99.22 John Doe: 3434.34 Ralph Smith: -19.08 Tom Smith: 123.22 Jane Baker: 1378.0			
(b)	The default capacity of HashMap is 16 and load factor is 0.75. What do you understand from this statement? The default capacity is 16. The load factor is between 0.0 and 1.0 determines how full the hashMap can be before it is resized upward When the number of elements is **greater than the capacity of the hash map multiplied by its load factor**, the hash Map is expanded. For constructors that do not take a fill ratio, **0.75** is used. Create a HashMap with the following entries and print out the keys and values. HashMap<Integer,String> hm = new HashMap<Integer,String>(); hm.put(400,"SE"); hm.put(100,"AJJ");hm.put(200,"CD"); hm.put(300,"OS"); [400=SE, 100=AJJ, 200=CD, 300=OS]	[05]	CO2	L3
2 (a)	Explain various methods available in Iterator and ListIterator and their behavior. **default void forEachRemaining(Consumer<? super E> action)** The action specified by action is executed on each unprocessed element in the collection. (*Added by JDK 8.)* boolean hasNext( ), E next() **hasNext():** Returns true if there are more elements. Otherwise, returns false. **next():** Returns the next element. Throws **boolean hasPrevious( ), E previous( )** **hasPrevious():** Returns true if there is a	[06]	CO2	L2

	previous element. Otherwise, returns false. **previous():** Returns the previous element. **default void remove( ), void add(E *obj*) remove:** Removes the current element. **Throws IllegalStateException** if an attempt is made to call remove( ) that is not preceded by a call to next( ). The default version throws an **UnsupportedOperationException.** **add:** Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**. **int nextIndex( ), int previousIndex( )** Returns the index of the next(nextIndex) or previous(previousIndex) element. **nextIndex():** If there is not a next element, returns the size of the list. **previousIndex() :** If there is not previous element, returns -1. **void set(E *obj*)** Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.			
(b)	PriorityQueue<String> pQue = new PriorityQueue<String>(); Collections.addAll(pQue, "Ruby","Cobol","Java","Python","PHP"); Iterator it = pQue.iterator(); **/* Write code to check and print all items in pQue using the iterator */** while(it.hasNext()){ System.out.println(it.next()); } System.out.println(); ArrayList<String> al = new ArrayList<String>(); Collections.addAll(al, "Ruby","Cobol","Java","Python","PHP"); ListIterator lt = al.listIterator(); System.out.println(lt.next()); System.out.println(lt.previousIndex()); System.out.println(lt.hasNext()); System.out.println(lt.previous()); Write the expected output for the above code snippet Ruby 0 true Ruby	[04]	CO2	L2
3 (a)	Write short notes on TreeSet, its constructors and it's behaviour. **creates a collection that uses a tree for storage** **Objects are stored in sorted, ascending order** **Access and retrieval times are quite fast.** **TreeSet( ) :** constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements	[06]	CO2	L2

	TreeSet(Collection<? extends E> c) : builds a tree set that contains the elements of c. **TreeSet(Comparator<? super E> comp) :** constructs an empty tree set that will be sorted according to the comparator specified by comp. **TreeSet(SortedSet<E> ss)** builds a tree set that contains the elements of ss. public static void main(String args[]) { // Create a tree set. TreeSet<String> ts = new TreeSet<String>(); // Add elements to the tree set. ts.add("C"); ts.add("A"); ts.add("B"); ts.add("E"); ts.add("F"); ts.add("D"); System.out.println(ts); System.out.println(ts.subSet("C", "F")); TreeSet<Integer> ts1 = new TreeSet<>(); ts1.add(12); ts1.add(8); ts1.add(18); ts1.add(5); ts1.add(11); ts1.add(17); ts1.add(4); System.out.println(ts1); //print a tail set System.out.println(ts1.tailSet(8)); } **Output:** [A, B, C, D, E, F] [C, D, E] [4, 5, 8, 11, 12, 17, 18] [8, 11, 12, 17, 18]			
(b)	Given a class Student with instance variables, usn, name, write code snippet to - Create 4 student objects. - Create a Treeset with comparator to sort by USN. - Add the four students to the Treeset - Display the tree set - Write the expected output. class Student{ String usn, name; public Student(String usn, String name){ this.usn=usn; this.name=name; } @Override public String toString() { return String.format("%s-%s", usn, name); } public String getusn(){ return usn; } } Answer: public class Main { public static void main(String[] args) { System.out.println("Hello World"); TreeSet<Student> ts = new TreeSet<Student>(Comparator.comparing(Student::getusn)); ts.add(new Student("cr100","Jane"));	[04]	CO2	L3

	ts.add(new Student("cr085","Bobby")); ts.add(new Student("cr040","Abby")); ts.add(new Student("cr025","Max")); ts.forEach(System.out::println); } }			
4 (a)	Method that returns an immutable set **unmodifiableSet()** Standard collections are not synchronized: **True** Collections method to return type safe view of Queue **checkedQueue()** Assume an ArrayList, **al** has [24,50,24,60,89,24,60] Collections method to count the number of time 24 appears **Collections.frequency(al,24)** Obtain a randomized list from al **Collections.shuffle(al)**	[3M]	CO2	L1
(b)	Explain any two legacy classes with code snippets. Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects. These classes are also known as Legacy classes. The following legacy classes defined by java.util package **Dictionary class:** Dictionary is an abstract class.It represents a key/value pair and operates much like Map. Dictionary is classified as obsolete, because it is fully superseded by Map class. With the advent of JDK 5, Dictionary was made generic. It is declared as shown here: class Dictionary<K, V> Here, K specifies the type of keys, and V specifies the type of values. 1. To add a key and a value, use the put() method. 2. Use get() to retrieve the value of a given key. Hashtable class: Hashtable stores key/value pair. However neither keys nor values can be null. Hashtable is synchronized while HashMap is not. Hashtable has following four constructors: Hashtable() //This is the default constructor. The default size is 11. Hashtable(int size) //This creates a hash table that has an initial size Hashtable(int size, float fillratio) Code snippet: import java.util.*; class HashTableDemo { public static void main(String args[]) { Hashtable<String,Integer> ht = new Hashtable<String,Integer>(); ht.put("a",new Integer(100)); ht.put("b",new Integer(200));	[07]	CO1	L2

```
  ht.put("c",new Integer(300));
  ht.put("d",new Integer(400));
  Set st = ht.entrySet();     //entrySet  returns a set containing Map.Entry values
  Iterator itr=st.iterator();
  while(itr.hasNext())
  {
    Map.Entry m=(Map.Entry)itr.next();
    System.out.println(itr.getKey()+" "+itr.getValue());
  }
 }
}
```

Vector
Implements List
-use it like ArrayList
Vector( ) : initial size of 10.
Vector(int *size*) :
Vector(int *size*, int *incr*)
incr specifies the number of elements to allocate each time that a vector is resized upward
By default it is ***doubled***.
Vector(Collection<? extends E> *c*)
int capacityIncrement; **// stores the increment** value
int elementCount; **// number of elements currently in the vector**
Object[ ] elementData; **//array that holds the vector**
Legacy Methods
void addElement(E el) – add element
E elementAt(int idx) – add at an index position
E firstElement( )
E lastElement()
int indexOf(Object o) and int **lastIndexOf(Object o)**.
boolean removeElement(Object o) or void **removeElementAt(int idx)**.

```
// Demonstrate various Vector operations.
import java.util.*;  class VectorDemo {
public static void main(String args[]) {
// initial size is 3, increment is 2  Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());  System.out.println("Initial capacity: " +
v.capacity());
v.addElement(1);     v.addElement(2);          v.addElement(3);     v.addElement(4);
System.out.println("Capacity after four additions: " +  v.capacity());
System.out.println("First element: " + v.firstElement());  System.out.println("Last
element: " + v.lastElement());  if(v.contains(3))
System.out.println("Vector contains 3.");
// Enumerate the elements in the vector.  Enumeration<Integer> vEnum =
v.elements();  System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
```

	System.out.print(vEnum.nextElement() + " "); // recommended to use for loop // System.out.println("\nElements in vector:"); //for(int i : v) //System.out.print(i + " "); System.out.println(); } } **Output:** Initial size: 0 Initial capacity: 3 Capacity after four additions: 5 First element: 1 Last element: 12 Vector contains 3. Elements in vector: 1 2 3 4 **E push(E element)** **E pop( )**. **E peek( ).** **EmptyStackException** is thrown if you call **pop( )** or **peek( )** when the invoking stack is empty. **boolean empty( )** :returns **true** if nothing is on the stack. **search( )** :determines whether an object exists on the stack returns the number of pops that are required to bring it to the top of the stack public class stackDemo { stackDemo() { Stack<Integer> st = new Stack<Integer>(); System.out.println("stack: " + st); st.push(42); st.push(66); st.push(99); st.pop();st.pop(); System.out.println("After popping two items: " + st); } } run: stack: [] After pushing 3 items: [42, 66, 99] After popping two items: [42]			
5 (a)	Explain different ways of creating strings in Java. Give code snippets. String s = new String(); • will create an instance of **String** with no characters in it String(char *chars*[ ]) • Creates a **String** initialized by an array of characters • char chars[] = { 'a', 'b', 'c' }; • String s = new String(chars); • String(char *chars*[ ], int *startIndex*, int *numChars*) • specify a subrange of a character array as an initializer • *startIndex* specifies the index at which the subrange begins • *numChars* specifies the number of characters to use char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' }; String s = new String(chars, 2, 3); //cde • String(String *strObj*)	[05]	CO3	L2

	– construct a **String** object that contains the same character sequence as another **String** object. – **s1** and **s2** contain the same string char c[] = {'J', 'a', 'v', 'a'}; String s1 = new String(c); String s2 = new String(s1); System.out.println(s1); System.out.println(s2); • Java's **char** type uses 16 bits to represent the basic Unicode character set. • strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. • 8-bit ASCII strings are common String(byte *chrs*[ ]) – *chrs* specifies the array of bytes String(byte *chrs*[ ], int *startIndex*, int *numChars*) – specify a subrange In each of these constructors – byte-to-character conversion is done by using the defaultcharacter encoding // Construct string from subset of char array. class SubStringCons { public static void main(String args[]) { byte ascii[] = {65, 66, 67, 68, 69, 70 }; String s1 = new String(ascii); System.out.println(s1); String s2 = new String(ascii, 2, 3); System.out.println(s2); } } ABCDEF CDE • String(StringBuffer *strBufObj*) – construct a **String** from a **StringBuffer** • **String** from a **StringBuilder** – construct a **String** from a **StringBuilder** • String(int *codePoints*[ ], int *startIndex*, int *numChars*) – extended Unicode character set – *codePoints* is an array that contains Unicode code points • There are also constructors that let you specify **Charset**			
(b)	Write a program using String methods to check if two strings are anagrams of each other, eg. Rescue and seCure are anagrams. String s1="Rescue"; String s2 ="seCure"; // check if anagram s1 = s1.toLowerCase(); s2 = s2.toLowerCase(); boolean isAnagram = true; if (s1.length() == s2.length()) { char[] s1Arr = s1.toCharArray(); Arrays.sort(s1Arr); char[] s2Arr = s2.toCharArray(); Arrays.sort(s2Arr);	[05]	CO3	L2

	System.out.println(Arrays.toString(s1Arr)); System.out.println(Arrays.toString(s2Arr)); for (int i=0; i<s1Arr.length;i++){ if(s1Arr[i] != s2Arr[i]){ isAnagram=false; break; } } }else isAnagram=false; if (isAnagram) System.out.println(s1 +" and "+s2+ " are anagrams"); else System.out.println(s1 +" and "+s2+ " are not anagrams"); Test Case 1 : Rescue, seCure Rescue and seCure are anagrams Test Case 2 : Bald, grow Bald and grow are not anagrams			
6 (a)	Explain various methods to check for occurrences of a substring or characters in a string. **Equals** • two strings for equality, use **equals( )**. • boolean equals(Object *str*) • *str* is the **String** object being compared with the invoking **String** object • **true** if the strings contain the same characters in the same order • **false** otherwise • ignores case differences, call **equalsIgnoreCase( )** • boolean equalsIgnoreCase(String *str*) • *str* is the **String** object being compared with the invoking **String** object • Return true or false in the same manner as equals() public class compDemo { public compDemo() { System.out.println("hello"); String s1 = "Hello"; String s2 = "Hello"; String s3 = "Good-bye"; String s4 = "HELLO"; System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));	[5]	CO3	L2

```
        System.out.println(s1 + " equals " + s3 + " -> " +
            s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " +
            s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
            s1.equalsIgnoreCase(s4));
    }

}
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO a-> false
Hello equalsIgnoreCase HELLO -> true
BUILD SUCCESSFUL (total time: 0 seconds)
```

regionMatches()

- compares a specific region inside a string with another specific region in another string.
- an overloaded form that allows you to ignore case in such comparisons
- boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- *startIndex* specifies the index at which the region begins within the invoking **String** object.
- **String** being compared is specified by *str2*
- The index at which the comparison will start within *str2* is specified by *str2StartIndex.*
- length of the substring being compared is passed in *numChars.*
- if *ignoreCase* is **true**, the case of the characters is ignored

```
// Region Matches
        System.out.println("\n** Region Matches **");
       s1 = "Java Programming";
       s2 = "programming";
       System.out.println(s1 + " equals " + s2 + " for region 5 to 10 -> " +
           s1.regionMatches(true, 5, s2,0,3));
** Region Matches **
Java Programming equals programming for region 5 to 10 -> true
```

- Specialized forms of regionMatches()
- **startsWith( )** method determines whether a given **String** begins with a specified string
- **endsWith( )** determines whether the **String** in question ends with a specified string.

	• boolean startsWith(String *str*) • boolean endsWith(String *str*) – *str* is the **String** being tested – If the string matches, **true** is returned – Otherwise, **false** is returned //Specialized form of region Matches System.out.println("\n** Specialized form of region Matches == **"); System.out.println("String 'Arrival' starts with 'Ar'->"+ "Arrival".startsWith("Ar")); System.out.println("String 'ended' ends with 'ed'->"+ "ended".endsWith("ed")); ** Specialized form of region Matches == ** String 'Arrival' starts with 'Ar'->true String 'ended' ends with 'ed'->true String 'Arrival' has 'rival' starting at index 2->true			
(b)	Given an ArrayList containing, [“KaYak”,”NooN”, “bargain”, “aNNa”,”rIoT”], write a program using String methods to determine if each string is a palindrome. Eg. Of palindrome, kaYak, nOoN aNNa. ArrayList<String> arrL = new ArrayList(); Collections.addAll(arrL, "KaYak","NooN", "bargain", "aNNa","rIoT"); for(String a: arrL){ String tmp = a.toLowerCase(); boolean isPalindrome = true; //System.out.println(tmp); for(int i=0, j=a.length()-1;i<=a.length()/2; i++, j--){ if(tmp.charAt(i)!=tmp.charAt(j)){ isPalindrome = false; break; } } if(isPalindrome) System.out.println(a+ " is a palindrome? "+ isPalindrome); } Output: KaYak is a palindrome? true NooN is a palindrome? true bargain is a palindrome? false aNNa is a palindrome? true rIoT is a palindrome? false	[5]	CO3	L3

CI CCI HoD

CO PO Mapping

CO-PO and CO-PSO Mapping																				
Course Outcomes		**Blooms Level**	**Modules covered**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**	**PSO4**
CO1	Interpret the need for advanced JAVA concepts like Enumerations, Wrapper Classes and Annotation.	**L1,L 2,L3**	**1**	2	2	2	0	3	0	0	0	3	0	0	0	3	0	0	2
CO2	Explain Collections Interface and Framework in development of modular applications.	**L1,L 2,L3**	**2**	2	2	2	0	3	0	0	0	3	0	0	0	3	0	0	2
CO3	Use In-Built String Handling Functions for development of Java Programs.	**L1,L 2,L3**	**3**	2	2	2	0	3	0	0	0	3	0	0	0	3	0	0	2
CO4	Describe how servlet fit into Java – Based web application architecture	**L1,L 2,L3**	**4**	3	2	2	0	3	0	0	0	3	0	0	0	3	0	0	3

CO5	Illustrate database access and details for managing information using the JDBC API	**L1,L 2,L3**	**5**	3	3	2	0	3	0	0	0	3	0	0	0	3	0	0	3

COGNITIVE LEVEL	REVISED BLOOMS TAXONOMY KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PROGRAM OUTCOMES (PO), PROGRAM SPECIFIC OUTCOMES (PSO)				CORRELATION LEVELS	
PO1	Engineering knowledge	PO7	Environment and sustainability	0	No Correlation
PO2	Problem analysis	PO8	Ethics	1	Slight/Low
PO3	Design/development of solutions	PO9	Individual and team work	2	Moderate/ Medium
PO4	Conduct investigations of complex problems	PO10	Communication	3	Substantial/ High
PO5	Modern tool usage	PO11	Project management and finance		
PO6	The Engineer and society	PO12	Life-long learning		
PSO1	Develop applications using different stacks of web and programming technologies				
PSO2	Design and develop secure, parallel, distributed, networked, and digital systems				
PSO3	Apply software engineering methods to design, develop, test and manage software systems.				
PSO4	Develop intelligent applications for business and industry				