Internal Assessment Test 1 – 7th September 2019										
Sub:	Advanced Java and J2EE					Sub Code:	17CS553	Branch:	CSE	
Date:	07/09/19	Duration:	90 mins	Max Marks:	50	Sem /Sec:	V A/B/C		OBE	

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1 (a)	**What are Enumerations? Explain values() and valueOf() methods for enumerations.** Answer: Enumeration: An enumeration is a list of named constants. In Java, an enumeration defines a class type. An enumeration can have constructors, methods, and instance variables. An enumeration can be defined simply by the keyword enum and by creating a list of enum variables. Eg: enum Subject //Enumeration defined { JAVA, CPP, C } The values( ) and valueOf( ) Methods: All enumerations automatically contain two predefined methods: values( ) and valueOf( ). Their general forms are : public static enum-type[ ] values( ) public static enum-type valueOf(String str) The values( ) method returns an array that contains a list of the enumeration constants. The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str. In both cases, enum-type is the type of the enumeration. //Program demonstrates the values( ) and valueOf( ) methods: // An enumeration of apple varieties. enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland } class EnumDemo { public static void main(String args[]) { Apple ap; System.out.println("Here are all Apple constants:"); // use values() Apple allapples[] = Apple.values(); for(Apple a : allapples) System.out.println(a); System.out.println(); // use valueOf() ap = Apple.valueOf("Winesap"); System.out.println("ap contains " + ap); } }	[05]	CO1	L2

(b)	Create an enumeration of type of any 4 restaurant menu items and their price as a variable. Define suitable constructors and method getPrice().Write driver code to demonstrate the enumeration. Answer: //Enumeration with Constructor, instance variable and Method enum Menu { FriedShrimp(10), BakedYam(9), FrenchFries(12), SteamedVeggies(15); // variable int price; Menu(int p) // Constructor { price = p; } //method int getPrice() { return price; } } public class EnumConstructor { public static void main(String[] args) { Menu ap; // Display price of FriedShrimp. System.out.println("FriedShrimp costs " + Menu.FriedShrimp.getPrice() + " cents.\n"); System.out.println(Menu.BakedYam.price); // Display all items and prices. System.out.println("All item prices:"); for(Menu a : Menu.values()) System.out.println(a + " costs " + a.getPrice() + " cents."); } }	[05]	CO1	L3
2a	Explain the use of ordinal(), compareTo() and equals() for enumerations with code snippets. Answer: Ordinal is a value that indicates an enumeration constant's position in the list of constants. It is retrieved by calling the ordinal( ) method, shown here: final int ordinal( ) It returns the ordinal value of the invoking constant. Ordinal values begin at zero. The ordinal value of two constants of the same enumeration can be compared by using the compareTo( ) method. It has this general form: final int compareTo(enum-type e) Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant. We can compare for equality an enumeration constant with any other object by using equals( ), which overrides the equals( ) method defined by Object	[05]	CO1	L2

```
// An enumeration of apple varieties.
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }
class EnumDemo {
public static void main(String args[]) {
Apple ap, ap2, ap3;
System.out.println("Here are all apple constants" + " and their ordinal values: ");
for(Apple a : Apple.values())
System.out.println(a + " " + a.ordinal());
ap =  Apple.RedDel;
ap2 = Apple.GoldenDel; ap3 = Apple.RedDel;
System.out.println();

 if(ap.compareTo(ap2) < 0)
System.out.println(ap + " comes before " + ap2);
if(ap.compareTo(ap3) == 0)
System.out.println(ap + " equals " + ap3);

if(ap.equals(ap3))
System.out.println(ap + " equals " + ap3);
if(ap == ap3)
System.out.println(ap + " == " + ap3);
}
```

2b	Write short notes on Type Wrappers with code snippets.	[05]	CO1	L2

Answer:

Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

Many standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these situations, type wrappers are used.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods.

i)Character

Character is a wrapper around a char. The constructor for Character is

Character(char ch)

Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

char charValue( )

It returns the encapsulated character.

```
// Code to demonstrate a type wrapper.
class Wrap
 {
public static void main(String args[])
{
Integer iOb = new Integer(100);
int i = iOb.intValue();
System.out.println(i + " " + iOb); // displays 100 100
}
```

3a	What is Auto-Boxing and Un-Boxing? Explain autoboxing in methods with code snippet. Answer: Autoboxing is the process by which a primitive type is automatically encapsulated(boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. Example: // Autoboxing/unboxing takes place with method parameters and return values. class AutoBox2 { // Take an Integer parameter and return // an int value; static int m(Integer v) { return v ; // auto-unbox to int } public static void main(String args[]) { // Pass an int to m() and assign the return value // to an Integer. Here, the argument 100 is autoboxed // into an Integer. The return value is also autoboxed // into an Integer. Integer iOb = m(100); System.out.println(iOb); } } This program displays the following result: 100 In the program, notice that m( ) specifies an Integer parameter and returns an int result. Inside main( ), m( ) is passed the value 100. Because m( ) is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.	[06]	CO1	L2
3b	What are Annotations? Explain @Override, @Inherited @Retention with code snippet. Answer: Annotations (Metadata) is a new facility added to Java that enables you to embed supplemental information into a source file. This information does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged @Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy. @Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass.	[04]	CO1	L2

Therefore, when a request for a specific annotation is made to the subclass,if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with @Inherited, then that annotation will be returned.

@Override is a marker annotation that can be used only on methods. A method annotated with @Override must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Java annotation example for @Override

```
public class MySuperClass {
public void doTheThing()
{
 System.out.println("Do the thing");
 } }
public class MySubClass extends MySuperClass{
@Override
public void doTheThing()
{
System.out.println("Do it differently");
}
 }
```

Java defines three retention policies, which are encapsulated within the java.lang.annotation.RetentionPolicy enumeration. They are SOURCE, CLASS, and RUNTIME.

An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of CLASS is stored in the .class file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time. Thus, RUNTIME retention offers the greatest annotation persistence.

Java annotation example for @Retention

The following version of MyAnno uses @Retention to specify the RUNTIME retention policy. So MyAnno will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
 @interface MyAnno {
String str();
int val();
 }
```

4a	Explain autoboxing/unboxing when used in an expression? In what situations are autoboxing recommended and in which situations should it be avoided. Answer: Within an expression, a numeric object is automatically unboxed.The outcome of the expression is reboxed, if necessary. //Program- Autoboxing/unboxing occurs inside expressions. class AutoBox { public static void main(String args[]) { Integer iOb, iOb2; int i; iOb = 100; System.out.println("Original value of iOb: " + iOb); // The following automatically unboxes iOb, performs the increment, then reboxes the result back into iOb. ++iOb; System.out.println("After ++iOb: " + iOb); // iOb is unboxed, the expression is evaluated, and result is reboxed assigned to iOb2. iOb2 = iOb + (iOb / 3); System.out.println("iOb2 after expression: " + iOb2); // The same expression is evaluated, but the result is not reboxed. i = iOb + (iOb / 3); System.out.println("i after expression: " + i); } } If Objects such as Integer or Double are used for abandoning primitives altogether, it is a very bad use of autoboxing/unboxing. For example, with autoboxing/unboxing, if we write: Double a, b, c; a = 10.0; b = 4.0; c = Math.sqrt(a*a + b*b); System.out.println("Hypotenuse is " + c); It is far less efficient than the equivalent code written using the primitive type double . Type wrappers should be used in only those cases in which an object representation of a primitive type is required, Eg.in Collections framework.	[05]	CO1	L2
4(b)	Create an annotation called info with author_name and version. Use it to annotate a method and obtain the values using reflection. Answer: import java.lang.annotation.*; import java.lang.reflect.*; @Retention(RetentionPolicy.RUNTIME) @interface info { String author_name(); double version(); } public class Test1 { // Annotate a method. @info(author_name = "Herbert Schildt", version = 1.0) public static void myMeth() { Test1 ob = new Test1(); // Obtain the annotation for this method and display the values of the members. try {	[05]	CO1	L3

	// First, get a Class object that represents this class. Class c = ob.getClass(); // Now, get a Method object that represents this method. Method m = c.getMethod("myMeth"); // Next, get the annotation for this class. info anno = m.getAnnotation(info.class); // Display the values. System.out.println(anno.author_name() + " " + anno.version()); } catch (NoSuchMethodException exc) { System.out.println("Method Not Found."); } } public static void main(String args[]) { myMeth(); } }			
5 (a)	Explain the core interfaces in the collection Framework. Answer: The interfaces that underpin collections are : i)Collection -Enables you to work with groups of objects; it is at the top of the collections hierarchy. ii)Deque - Extends Queue to handle a double-ended queue. (Added by Java SE 6.) iii)List - Extends Collection to handle sequences (lists of objects). iv)NavigableSet- Extends SortedSet to handle retrieval of elements based on closest-match searches. v)Queue -Extends Collection to handle special types of lists in which elements are removed only from the head. vi)Set- Extends Collection to handle sets, which must contain unique elements. vi)- SortedSet Extends Set to handle sorted sets. Collection is a generic interface that has this declaration: interface Collection<E> Here, E specifies the type of objects that the collection will hold. Objects are added to a collection by calling add( ).You can add the entire contents of one collection to another by calling addAll( ). You can remove an object by using remove( ). To remove a group of objects, call removeAll( ). You can remove all elements except those of a specified group by calling retainAll( ). To empty a collection, call clear( ). You can determine whether a collection contains a specific object by calling contains( ). To determine whether one collection contains all the members of another, call containsAll(). We determine when a collection is empty by calling isEmpty().The number of elements currently held in a collection can be determined by calling size( ). The toArray() methods return an array that contains the elements stored in the invoking collection. The first returns an array of Object. ii)List is a generic interface that has this declaration: interface List<E> Here, E specifies the type of objects that the list will hold.	[05]	CO2	L1

	iii) The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements. So the add() method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own. Set is a generic interface that has this declaration: interface Set<E> Here, E specifies the type of objects that the set will hold. iv) The Queue Interface The Queue interface extends Collection and declares the behavior of a queue, which is a first-in, first-out list. Queue is a generic interface that has this declaration: interface Queue<E> Some methods are: E element( ) Returns the element at the head of the queue. The element is not removed. boolean offer(E obj) Attempts to add obj to the queue. Returns true if obj was added and false otherwise. E peek( ) Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed. E poll( ) Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty. E remove( ) Removes the element at the head of the queue, returning the element in the process.			
5(b)	List any 6 methods with the method signature and its purpose from the collection interface and any exceptions thrown. Answer: 1.boolean add(E obj) Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection and the collection does not allow duplicates. 2.boolean addAll(Collection<? extends E> Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false. 3.void clear( ) Removes all elements from the invoking collection. 4.boolean contains(Object obj) Returns true if obj is an element of the invoking collection. Otherwise, returns false. 5. boolean containsAll(Collection<?> c) Returns true if the invoking collection contains all elements of c. Otherwise, returns false. 6.boolean equals(Object obj) Returns true if the invoking collection and obj are equal. Otherwise, returns false.	[05]	CO2	L2

	The Exceptions thrown are: ✓ A ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. ✓ A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the collection. ✓ An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length collection that is full.			
6(a)	Write short notes on ArrayList. Answer: The ArrayList class extends AbstractList and implements the List interface. ArrayList is a generic class that has this declaration: class ArrayList<E> Here, E specifies the type of objects that the list will hold. ArrayList supports dynamic arrays that can grow as needed. ArrayList has the constructors shown here: ArrayList( ) ArrayList(Collection<? extends E> c) ArrayList(int capacity) Methods in Java ArrayList: 1.retainAll(Collection c): Retains only the elements in this list that are contained in the specified collection. 2.contains(Object o): Returns true if this list contains the specified element. 3.remove(int index): Removes the element at the specified position in this list. 4.remove(Object o): Removes the first occurrence of the specified element from this list, if it is present. 5.get(int index): Returns the element at the specified position in this list. 6.subList(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. 7.set(int index, E element): Replaces the element at the specified position in this list with the specified element. 8.size(): Returns the number of elements in this list. 9. removeAll(Collection c): Removes from this list all of its elements that are contained in the specified collection. 10. ensureCapacity(int minCapacity): Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. 11. listIterator(): Returns a list iterator over the elements in this list (in proper sequence). 12. listIterator(int index): Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.	[05]	CO2	L1

	13.isEmpty(): Returns true if this list contains no elements. 14.removeRange(int fromIndex, int toIndex): Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. 15.void clear(): This method is used to remove all the elements from any list. 16.void add(int index, Object element): This method is used to insert a specific element at a specific position index in a list. 17.void trimToSize(): This method is used to trim the capacity of the instance of the ArrayLis to the list's current size. 18.int indexOf(Object O)-Finds index of given object Eg: import java.util.*; class ArrayListDemo { public static void main(String args[]) { // Create an array list. ArrayList<String> al = new ArrayList<String>(); System.out.println("Initial size of al: " + al.size()); al.add("C"); al.add("A"); al.add("E"); al.add("B"); al.add("D"); al.add(1, "A2"); System.out.println("Size of al after additions: " + al.size()); System.out.println("Contents of al: " + al); // Remove elements from the array list. al.remove("F"); al.remove(2); System.out.println("Size of al after deletions: " + al.size()); System.out.println("Contents of al: " + al); } }			
6 (b)	Write a program to initialize an ArrayList with 5 Integer objects. Calculate and display the sum and average of the items in the list. Answer: import java.util.*; public class Test1 { public static void main(String[] args) { ArrayList<Integer> list=new ArrayList<Integer>(); list.add(10);list.add(20);list.add(30); list.add(40); list.add(50); double sum = 0; for (int i : list) { sum += i; } double average = sum / list.size(); System.out.println("Average = " + average); } }	[05]	CO2	L3

7(a)	Explain the constructors for TreeSet. Write a java program to create TreeSet collection and access via an Iterator. Answer: TreeSet extends AbstractSet and implements the NavigableSet interface. TreeSet has the following constructors: TreeSet( ) TreeSet(Collection<? extends E> c) TreeSet(Comparator<? super E> comp) TreeSet(SortedSet<E> ss) Example that demonstrates a TreeSet: import java.util.*; class TreeSetDemo { 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); } } The output from this program is shown here: [A, B, C, D, E, F] As explained, because TreeSet stores its elements in a tree, they are automatically arranged in sorted order.	[05]	CO2	L3

7(b)	Explain any 2 legacy classes of Java's collection Framework. Answer: 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 ❑ HashTable 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)); 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()); } } }	[05]	CO2	L1
8(a)	Write short notes on all the methods defined by the SortedSet interface. Answer: The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration: interface SortedSet<E> Here, E specifies the type of objects that the set will hold.	[06]	CO2	L1

	Methods defined by the SortedSet interface are: 1.Comparator<? super E> comparator( ) Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned. 2.E first( ) Returns the first element in the invoking sorted set. 3.SortedSet<E> headSet(E end) Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set. 4.E last( ) Returns the last element in the invoking sorted set. 5.SortedSet<E> subSet(E start , E end) Returns a SortedSet that includes those elements between start and end– 1. Elements in the returned collection are also referenced by the invoking object. 6.SortedSet<E> tailSet(E start) Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.			
8 (b)	Write a Java program to demonstrate ArrayDeque by using it to create a stack Answer: The following program demonstrates ArrayDeque by using it to create a stack: // Demonstrate ArrayDeque. import java.util.*; class ArrayDequeDemo { public static void main(String args[]) { // Create a tree set. ArrayDeque<String> adq = new ArrayDeque<String>(); // Use an ArrayDeque like a stack. adq.push("A"); adq.push("B"); adq.push("D"); adq.push("E"); adq.push("F"); System.out.print("Popping the stack: "); while(adq.peek() != null) System.out.print(adq.pop() + " "); System.out.println(); } } The output is shown here: Popping the stack: F E D B A	[04]	CO2	L3