USN | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 3 – Nov. 2018

Sub:	Programming in JAVA				Sub Code:	15CS561	Branch:	ECE/EEE
Date:	22-11-2018	Duration:	90 min's	Max Marks: 50	Sem / Sec:	5 (ALL SEC)		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Define Interface. Explain how to define, implement and assign variable in interface to perform "one interface, multiple methods".	[2+8]	CO4	L3
2a)	Write short note on i) importing package ii) Accesses protection	[5] [5]	CO3, CO4	L2
b)	What is the role of interface while implementing multiple inheritance in java?		CO4	L3
3a)	Write a program which contains one method which will throw "IllegalAcessException" and use proper exception handles so that exception should be printed.	[5]	CO2, CO4	L3
b)	Demonstrate the working of Nested try block with an example.	[5]	CO4	L3
4 a)	What are chained Exceptions? Write a JAVA program to demonstrate the concept of chained exceptions.	[5]	CO2, CO4	L3
b)	What is package? Write the syntax of defining and executing package.	[5]	CO3, CO4	L2
5	What are Applets? Explain different stages in the lifecycle of Applet.	[10]	CO5	L2

USN | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 3 – Nov. 2018

Sub:	Programming in JAVA				Sub Code:	15CS561	Branch:	ECE/EEE
Date:	22-11-2018	Duration:	90 min's	Max Marks: 50	Sem / Sec:	5 (ALL SEC)		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Define Interface. Explain how to define, implement and assign variable in interface to perform "one interface, multiple methods".	[2+8]	CO4	L3
2a)	Write short note on i) importing package ii) Accesses protection	[5] [5]	CO3, CO4	L2
b)	What is the role of interface while implementing multiple inheritance in java?		CO4	L3
3a)	Write a program which contains one method which will throw "IllegalAcessException" and use proper exception handles so that exception should be printed.	[5]	CO2, CO4	L3
b)	Demonstrate the working of Nested try block with an example.	[5]	CO4	L3
4 a)	What are chained Exceptions? Write a JAVA program to demonstrate the concept of chained exceptions.	[5]	CO2, CO4	L3
b)	What is package? Write the syntax of defining and executing package.	[5]	CO3, CO4	L2
5	What are Applets? Explain different stages in the lifecycle of Applet.	[10]	CO5	L2

6	What is an Enum? Write a JAVA program to demonstrate values() and valueof() methods.	[2+8]	CO3	L3
7a)	Write an applet program to display the message “CMRIT”. Set background and foreground to Cyan and Red respectively.	[5]	CO5	L3
b)	What is boxing and unboxing? Explain with example.	[5]	CO4	L2
8 (a)	What is difference between String, StringBuffer and StringBuilder class in JAVA?	[5]	CO3	L2
(b)	Explain the String searching functions with example. .	[5]	CO3	L3

6	What is an Enum? Write a JAVA program to demonstate values() and valueof() methods	[2+8]	CO3	L3
7a)	Write an applet program to display the message “CMRIT”. Set background and foreground to Cyan and Red respectively.	[5]	CO5	L3
b)	What is boxing and unboxing? Explain with example.	[5]	CO4	L2
8 (a)	What is diffrence between String, StringBuffer and StringBuilder class in JAVA?	[5]	CO3	L2
(b)	Explain the String searching functions with example.	[5]	CO3	L3

1) Define Interface. Explain how to define, implement and assign variable in interface to perform "one interface, multiple methods". [2+8]

Defining Interface:

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
 return-type method-name1(parameter-list);
 return-type method-name2(parameter-list);
 type final-varname1 = value;
 type final-varname2 = value;
 // ...
 return-type method-nameN(parameter-list);
 type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code.

Here is an example of an interface definition.

```
  Interface in1{
    final int a=3;
   Void display();
 }
```

Implementing Interface

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {
  // class-body
   }
```

Here two classes class A and class B has implemented interface in1 and they have defined their own version of display with the help of final variable which is defined in the interface.

```
Class A implements in1{
  int b;
 Void display(){
  b=a+2;
  System.ot.println("B is "+b);
}}
```

```
Class B implements in1{
    int c;
    void display(){
        b=a+2;
       System.ot.println("B is "+b);
   }}
```

So by the above example we have implemented perform "one interface, multiople methods".

2)a) Write short note on [5]
i) importing package
ii) Accesses protection
b)What is the role of interface while implementing multiple inheritance in java? [5]

a) i) import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

1. **Using fully qualified name** (But this is not a good practice.)
   If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface, which can look a little untidy if the package name is long.
   This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.
2. **To import only the class/classes you want to use**

If you import packagename.classname then only the class with name classname in the package with name packagename will be available for use.

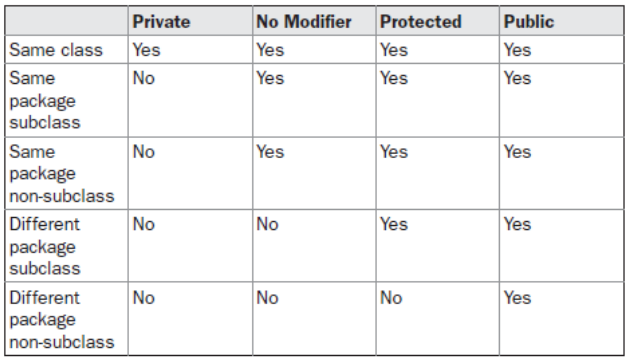
3. **To import all the classes from a particular package**

If you use packagename.*, then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use.
The import keyword is used to make the classes and interface of another package accessible to the current package.

Access Protection

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

• Subclasses in the same package
• Non-subclasses in the same package
• Subclasses in different packages
• Classes that are neither in the same package nor subclasses
The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. Following sums up the interactions. Class member access can be summarized with the following table



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

b) **Multiple Inheritance** allows a class to have more than one super class and to inherit features from all parent class. it is achieved using interface. A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. Following is an example of multiple inheritance implemented by interface concepts

```
nterface vehicleone{

        int  speed=90;

        public void distance();

}

interface vehicletwo{

        int distance=100;

        public void speed();

}

class Vehicle  implements vehicleone,vehicletwo{

        public void distance(){

                        int  distance=speed*100;

                        System.out.println("distance travelled is "+distance);
        }

        public void speed(){

                        int speed=distance/100;

        }

}

class MultipleInheritanceUsingInterface{

        public static void main(String args[]){

                        System.out.println("Vehicle");

                        obj.distance();

                        obj.speed();                }

}

Output is:

distance travelled is 9000
```

3) a) Write a program which contains one method which will throw "IllegalAcessException" and use proper exception handles so that exception should be printed. [5]
b) Demonstrate the working of Nested try block with an example. [5]

a)

```
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
                }
        public static void main(String args[]) {
          try {
              throwOne();
          } catch (IllegalAccessException e) {
                System.out.println("Caught " + e);
           }
        }
  }
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

b) The try statement can be nested. That is, a try statement can be inside the block of another try.

- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception

```
class NestTry {
    public static void main(String args[]) {
     try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
         try {
            if(a==1)
                    a = a/(a-a);
            if(a==2) {
               int c[] = { 1 };
               c[42] = 99;
```

```
                    }
               }
          catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Array index out-of-bounds: " + e);
          }
               }
 catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
                         }
               }
     }
```

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block.

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
```

Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled.

```
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
```

If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

```
C:\>java NestTry One Two
a = 2
```

- Array index out-of-bounds:
- java.lang.ArrayIndexOutOfBoundsException:42

4) a) What are chained Exceptions? Write a JAVA program to demonstrate the concept of chained exceptions.
b) What is package? Write the syntax of defining and executing package. [5+5]

a) The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

- Throwable(Throwable *causeExc*)
- Throwable(String *msg*, Throwable *causeExc*)

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception
The chained exception methods added to **Throwable** are **getCause( )** and **initCause( ).**

- Throwable getCause( )
- Throwable initCause(Throwable causeExc)

The **getCause( )** method returns the exception that underlies the current exception. If there is no underlying exception, null is returned. The **initCause( )** method associates ***causeExc*** with the invoking exception and returns a reference to the exception.
Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
   class ChainExcDemo {
      static void demoproc() {
         NullPointerException e = new NullPointerException("top layer");
         e.initCause(new ArithmeticException("cause"));
         throw e;
            }
       public static void main(String args[]) {
       try {
        demoproc();
      } catch(NullPointerException e) {
          System.out.println("Caught: " + e);
          System.out.println("Original cause: " + e.getCause());
        }
      }
    }
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

b) Packages

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. We can define classes inside a package that are not accessible by code outside that package. We can also define class members that are only exposed to other members of the same package. This allows us classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If we omit the package statement, the class names are put into the default **package**, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, we will define a **package** for our code.
This is the general form of the package statement:

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the .class files for any classes we declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.
We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package java.awt.image; needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored

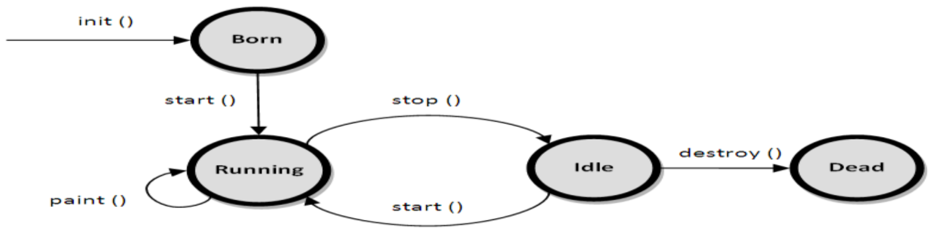
For Executing a package :
We can just import the package before our class definition where we want to include the classes in the corresponding package by the line:

```
import packagename;
```

5) What are Applets? Explain different stages in the lifecycle of Applet [10]

- a) Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

b)The life cycle of an applet is as shown in the figure below:



As shown in the above diagram, the life cycle of an applet starts with *init()*method and ends with *destroy()* method. Other life cycle methods are *start(), stop()* and *paint()*. The methods to execute only once in the applet life cycle are *init()* and *destroy()*. Other methods execute multiple times.

init(): The init() method is the first method to execute when the applet is executed. Variable declaration and initialization operations are performed in this method.

start(): The start() method contains the actual code of the applet that should run. The start() method executes immediately after the *init()* method. It also executes whenever the applet is restored, maximized or moving from one tab to another tab in the browser.

stop(): The stop() method stops the execution of the applet. The stop() method executes when the applet is minimized or when moving from one tab to another in the browser.

destroy(): The destroy() method executes when the applet window is closed or when the tab containing the webpage is closed. *stop()* method executes just before when destroy() method is invoked. The destroy() method removes the applet object from memory.

paint(): The paint() method is used to redraw the output on the applet display area. The paint() method executes after the execution of *start()* method and whenever the applet or browser is resized.

The method execution sequence when an applet is executed is:
init()   start()   paint()
The method execution sequence when an applet is closed is:
stop()   destroy()

6) What is an Enum? Write a JAVA program to demonstate values() and valueof() methods [2+8]
Enum

- An enumeration is a list of named constants.
- Final variables previously provide somewhat similar functionality.
- Beginning with JDK 5, enumerations were added to the Java language, and they are now available to
- the Java programmer.
- In C++, enumerations are simply lists of named integer constants but in Java, an enumeration defines a class type.
- In Java, an enumeration can have constructors, methods, and instance variables.

Enumeration Fundamentals

An enumeration is created using the enum keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
        enum Apple {
          Jonathan, GoldenDel, RedDel, Winesap, Cortland
          }
```

The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants. Each is implicitly declared as a public, static final member of Apple.
Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case.

Values() and valuesof()

- All enumerations automatically contain two predefined methods: **values( )** and **valueOf( ).**
- Their general forms are shown here:

  public static enum-type[ ] values( ) // returns an array that contains a list of the enumeration constants.

  public static enum-type valueOf(String str) // returns the enumeration constant whose value corresponds to the string passed in ***str***.
- In both cases, enum-type is the type of the enumeration.

For example, in the case the return type of Apple.valueOf("Winesap") is Winesap.

```
   enum Apple {
        Jonathan, GoldenDel, RedDel, Winesap, Cortland
             }
class EnumDemo2 {
       public static void main(String args[])   {
      Apple ap;
      System.out.println("Here are all Apple constants:");
      Apple allapples[] = Apple.values();
      for(Apple a : allapples)
          System.out.println(a);
     System.out.println();
     ap = Apple.valueOf("Winesap");
    System.out.println("ap contains " + ap);  }  }
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland
ap contains Winesap
```

7) a) **Write an applet program to display the message "CMRIT". Set background and foreground to Cyan and Red respectively.**
b) What is boxing and unboxing? Explain with example [5+5]

a) MyApplet_CMRIT.java

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class MyApplet_CMRIT extends Applet
{
public void init()
{
setBackground(Color.cyan);
setForeground(Color.red);
}
public void paint(Graphics g)
{
g.drawString("CMRIT", 100, 200);
}
}
```

MyApplet_CMRIT_HTML.html

```
<html>
<applet code = "MyApplet_CMRIT.class" width=400 height=400>
</applet>
</html>
```

Compiling: javac MyApplet_CMRIT.java
Executing: appletviewer MyApplet_CMRIT_HTML.html

b) Demonstrate a Boxing and Unboxing.

```
 class Wrap {
            public static void main(String args[]) {
              Integer iOb = new Integer(100); //The process of encapsulating a value within an object is called boxing
              int i = iOb.intValue();   //The process of extracting a value from a type wrapper is called unboxing
              System.out.println(i + " " + iOb); //          displays 100 100
```

} }

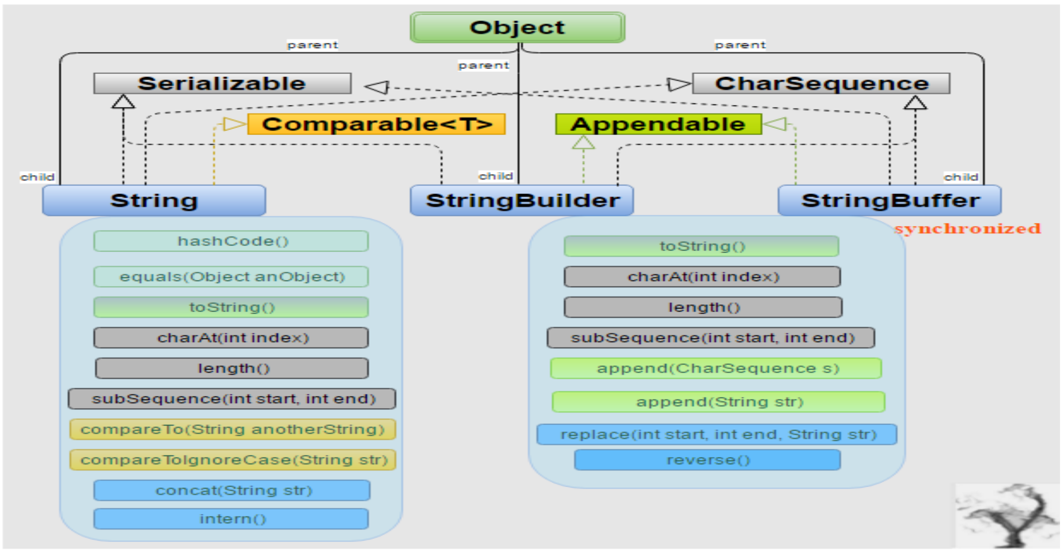
8) a) What is diffrence between String, StringBuffer and StringBuilder class in JAVA? [5]
b) Explain the String searching functions with example [5]

Mutability Difference:
String is **immutable**, if you try to alter their values, another object gets created,
whereas StringBuffer and StringBuilder are **mutable** so they can change their values

Thread-Safety Difference:
The difference between StringBuffer and StringBuilder is that StringBuffer is thread-safe. So when the application needs to be run only in a single thread then it is better to use StringBuilder. StringBuilder is more efficient than StringBuffer. StringBuffer is thread safe and synchronized whereas StringBuilder is not, thats why StringBuilder is more faster than StringBuffer

The Basics:
String is an immutable class, it can't be changed. StringBuilder is a mutable class that can be appended to, characters replaced or removed and ultimately converted to a String StringBufferis the original synchronized version of StringBuilder

String concat + operator internally uses StringBuffer or StringBuilder class.



b) The **String** class provides two methods that allow you to search a string for a specified character or substring:

• **indexOf( )** Searches for the first occurrence of a character or substring.
• **lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

To search for the first occurrence of a character, use
int indexOf(int *ch*)

To search for the last occurrence of a character, use
int lastIndexOf(int *ch*)
Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use
int indexOf(String *str*)
int lastIndexOf(String *str*)
Here, *str* specifies the substring.

We can specify a starting point for the search using these forms:
int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)
int indexOf(String *str*, int *startIndex*)

```
int lastIndexOf(String str, int startIndex)
```

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

```
// Demonstrate indexOf() and lastIndexOf().
    class indexOfDemo {
      public static void main(String args[]) {
      String s = "Now is the time for all good men " + "to come to the aid of their country.";
      System.out.println(s);
      System.out.println("indexOf(t) = " + s.indexOf('t'));
      System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
      System.out.println("indexOf(the) = " + s.indexOf("the"));
      System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
      System.out.println("indexOf(t, 10) = " +s.indexOf('t', 10));
 System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
 System.out.println("indexOf(the, 10) = " +s.indexOf("the", 10));
 System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the", 60));
}
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```