Name: Bilal Khondannavar **Class:** BCA V

Internet Programming - 2016

1.a) Explain the general program structure of Java. (2017- 2A)

b) Explain the features of Java. (2017- 2B)

2.(A) What are command line arguments? Explain with an example.

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in Java programs by using what are known as command line arguments. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution. It may be recalled that was invoked for execution at the command line as follows:

```
java Test
```

Here, we have not supplied any command line arguments. Even if we supply arguments, the program does not know what to do with them.

We can write Java programs that can receive and use the arguments provided in the command line. Recall the signature of the main( ) method used in our earlier example programs:

```
public static void main (String args[ ])
```

As pointed out earlier, args is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array args as its elements. We can simply access the array elements and use them in the program as we wish. For example, consider the command line

```
java Test BASIC FORTRAN C++ Java
```

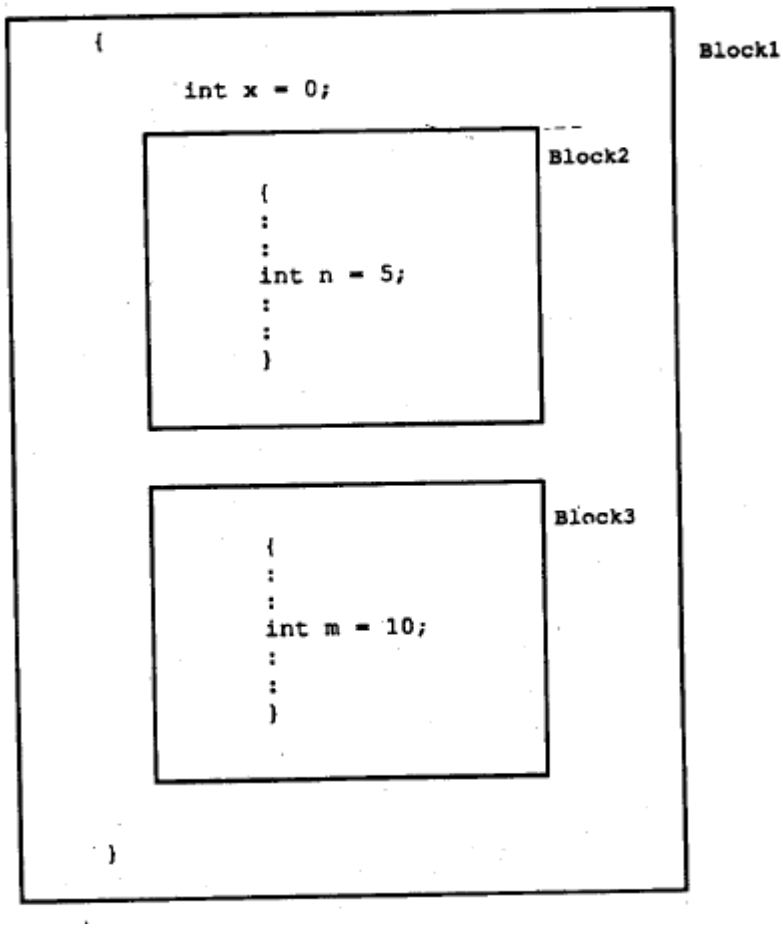
```
/*
* This program uses command line
* arguments as input.
*/
Class ComLineTest
{
public static void main(String args[ ])
{
int count, i=O;
String string;
count = args.length;
System.out.println("Number of arguments = " + count);
while (i < count)
{
string = args[i];
i = i + 1;
System.out.println(i+ " : " + "Java is " +string+ "!");
}
```

```
}
}
```

java **ComLineTest** Simple Object Oriented Distributed Robust Secure
Portable. Multithreaded Dynamic Upon execution, the command line arguments Simple, Object_Oriented, etc. are passed to the program through the array args as discussed earlier. That i& the element args[0] contains Simple, args[1] contains Object_Oriented, and SO on. These elements are accessed using the loop variable i as an index like
name = args[i]
The index i is incremented using a while loop until all the arguments are accessed. The number of arguments is obtained by statement:
count = args.length;
The output of the program would be as follows:
Number of arguments = 8
1 : Java is Simple!
2 : Java is Object_Oriented!
3 : Java is Distributed!
4 : Java is Robust!
5 : Java is Secure!
6 : Java is Portable!
7 : Java is Multithreaded!
8 : Java is Dynamic!
Note how the output statement concatenates the strings while printing.

2.(b) Explain scope of variables in java with example.

Java variables are actually classified into three kinds:
• *instance* variables,
• *class* variables, and
• *local* variables.
Instance and class variables are declared inside a class. Instance variables are created when the objects are instantiated' and therefore they are associated with the objects. They take different *values* for each object. On the other hand, class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable. Variables declared and used inside methods are called *local .variables.* They are called so because they are not available for use outside the method definition. Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible (i.e., usable) is called its *scope.*
We can have program blocks within other program blocks (called nesting) as shown in Figure. Each block can contain its own set of local variable declarations. We cannot, however, *declare* a variable to have the same name as one in an outer block. In Fig., the variable x declared in Block1 is available in all the three blocks. However, the variable n declared in Block2 is available only in Block2, because it goes out of the scope at the end of Block2. Similarly, m is accessible only in Block1



2.(c) Write the difference between constructor and method.

No.	Methods	Constructors
1	It is not necessary to use the same name of the class to create a Method.	The name of the Constructor should be same as the class name it resides.
2	Method is an ordinary member function which is used to expose the behavior of an object	Constructor is a member function of a class used to initialize the state of an object
3	Methods must have return type unless it is specified as void.	Constructor does not have a return type.
4	Compiler does not create a method in any case if one is not available.	Java compiler creates default constructor if the program doesn't have one.
5	Methods invoked explicitly. Invoked using the dot operator.	Constructor invoked implicitly. i. e. Invoked using the keyword 'new'.

3.(a) What is constructor? Explain overloaded constructor with an example.

The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like getData to initialize each object individually using statements like,

```
rectl.getData(15,lO);
```

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a *constructor*, that enables an object to initialize itself

when it is created. Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even void. This is because they return the instance of the class itself. Let us consider our Rectangle class again. We can now replace the getData method by a constructor method as shown below:

```
class Rectangle
{
int length ;
int width ;
Rectangle(int x, int y) / / Constructor method
{
length = x ;
width = Y ;
}
int rectArea( )
{
r~turn(length * width);
}
}
```

3.(b)Explain the looping statements used in java with example.

The While Statement

The while is an *entry-controlled* loop statement. The *test* condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of' the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

The basic format of the while statement is:

Initialization;

While(teat condition)
{
Body of the loop
}

Consider the following code segment:

```
..... ..... .
sum = 0;
n = 1;
while (n <= 10)
{
sum = sum + n * n;
n = n+1;
}
System.out.prntln("Sum = "+ sum);
..... .
```

The body of the loop is executed 10 times for n = 1,2, ... , 10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n< 11,-the-result would be the same.

The Do statement:

The **while** loop construct that we have discussed in the previous section makes a test condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the form:

Initialization
Do
{
Body of the loop
}
While(test condition);

On reaching the do statement, the program proceeds to ~valuate the body of the loop first. At the end of the loop, the *test condition* in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement. Since the *test condition* is evaluated at the bottom of the loop, the do ••••while construct provides an *exit control1ed* loop and therefore the body of the loop is always executed at least once.

Consider an example:

```
·..'.....
·.....
 ·i = 1;
sum = 0;
```

```
do
{
sum = sum + i;

}
while(sum < 40 || i < 10);
```
.....
The loop will be executed as long as one of the two relations is true.

THE FOR STATEMENT

The for loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the for loop is

```
for(initialization; test condition; increment)
{
        Body of the loop
}
```

Consider the following segment of a program

```
for (x = 0 ; x < = 9 ; x = x+l)
{
System.out.println(x);
}
```

This for loop is executed 10 times and prints the digits 0 to 9 in a vertical line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the increment section, x = x+ 1.

The for statement allows for negative increments. For example, the loop discussed above can be written as follows:

```
for. ( x = 9 ; x > ~ 0 ; x = x-l)
System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

4.(a) Explain Bitwise operators with an example.

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of 'bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double.

& bitwise AND
! bitwise OR
A bitwise exclusive OR - one's complement
« shift left
» shift right
»> shift right with zero fill

Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

Example 1: Bitwise OR

```
class BitwiseOR {
   public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        result = number1 | number2;
        System.out.println(result);
   }
}
```

When you run the program, the output will be:

29

Bitwise AND

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.

The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0. For example,

```
class BitwiseAND {
   public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        result = number1 & number2;
        System.out.println(result);
   }
}
```

When you run the program, the output will be:

```
8
```

Bitwise Complement

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and

```
class Complement {
   public static void main(String[] args) {

        int number = 35, result;

        result = ~number;
        System.out.println(result);
   }
}
```

When you run the program, the output will be:

```
-36
```

Bitwise XOR

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0. For example,

Example 4: Bitwise XOR

```
class Xor {
   public static void main(String[] args) {
```

```
        int number1 = 12, number2 = 25, result;

        result = number1 ^ number2;
        System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
21
```

Signed Left Shift

The left shift operator `<<` shifts a bit pattern to the left by certain number of specified bits, and zero bits are shifted into the low-order positions.

Example 5: Signed Left Shift

```
class LeftShift {
    public static void main(String[] args) {

        int number = 212, result;

        System.out.println(number << 1);
        System.out.println(number << 0);
        System.out.println(number << 4);
    }
}
```

When you run the program, the output will be:

```
424

212

3392
```

Signed Right Shift

The right shift operator `>>` shifts a bit pattern to the right by certain number of specified bits.

Example 6: Signed Right Shift

```
class RightShift {
   public static void main(String[] args) {

        int number = 212, result;

        System.out.println(number >> 1);
        System.out.println(number >> 0);
        System.out.println(number >> 8);
   }
}
```

When you run the program, the output will be:

```
106

212

0
```

Unsigned Right Shift

The unsigned right shift operator `<<` shifts zero into the leftmost position.

Example 7: Signed and UnSigned Right Shift

```
class RightShift {
   public static void main(String[] args) {

        int number1 = 5, number2 = -5;
```

```
        // Signed right shift
        System.out.println(number1 >> 1);

        // Unsigned right shift
        System.out.println(number1 >>> 1);

        // Signed right shift
        System.out.println(number2 >> 1);

        // Unsigned right shift
        System.out.println(number2 >>> 1);
    }
}
```

When you run the program, the output will be:

```
2

2

-3
```

2147483645

Notice, how signed and unsigned right shift works differently for 2's complement.

The 2's complement of 2147483645 is 3.

4.(b) Differentiate between method overloading and method overriding.

	Method overloading	Method overriding
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different	In Method Overriding, sub class have the same method with same name and exactly the same

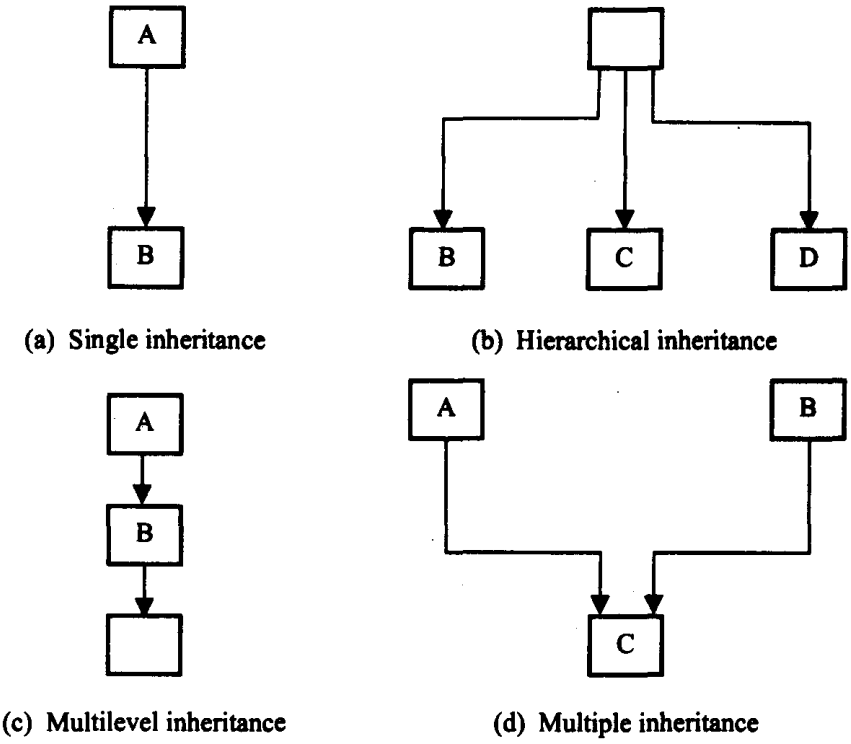
	number of parameters or parameters having different types and order.	number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.
	Overloading and Overriding is a kind of polymorphism. Polymorphism means "one name, many forms".	
Polymorphism	It is a **compile time polymorphism**.	It is a **run time polymorphism**.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have **different signature**.	In Method Overriding, methods must have **same signature**.
Relationship of Methods	In Method Overloading, relationship is there between methods of same class.	In Method Overriding, relationship is there between methods of super class and sub class.
Criteria	In Method Overloading, methods have same name different signatures but in the same class.	In Method Overriding, methods have same name and same signature but in the different class.
No. of Classes	Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

5.(a) Define inheritance. Explain different types of inheritance in java.

The old class is known as the *base class* or *super class* or *parent class* and the new one is called the *subclass* or *derived class* or *child class.* The inheritance allows subclasses to inherit all the variables and methods of their parent classes. Inheritance may take different forms:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)

Fig. 8.3



1. Single Inheritance

Single Inheritance is the simple inheritance of all, When a class extends another class(Only one class) then we call it as **Single inheritance. Class B** extends only one class **Class A.** Here **Class B** will be the **Sub class** and **Class A** will be one and only **Super class**.

```
class Room
{
int length;
int breadth;
Room(int' x, int y)
{
length = X;
breadth = y;
}
int area ( )
{
return (length *breadth);
}
}
class BedRoom extends Room // Inheriting Room
{
int height;
BedRoom(int x, int y, int x)
{
8uper(x,y); // pass values to superclass
height = z;
```

```
}
int volume ( )
{
return (length *breadth *height);
}
}
class InherTest
{
public static void main (String args[ ])
{
BedRoom rooml = new BedRoom(14,12,10);
int areal =rooml.area( ); // superclass method
int volumel = rooml.volume( ) ; // baseclass method
System;out.println("Areal = "+ areal);
System.out.println("Volumel = "+ volumel);
}
}
```

The output of Program 8.5 is:

```
Areal = 168
Volumel = 1680
```

The program defines a class Room and extends it to another class BedRoom. Note that the class BedRoom defines its own data members and methods. The subclass BedRoom now includes three instance variables, namely, length, breadth and height and two methods, area and volume. The constructor in the derived class uses the super keyword to pass values that are required by the base constructor. The statement

`BedRoom rooml = new BedRoom(14,12,10);`

calls first the BedRoom constructor method, which in turn calls the Room constructor method by using the super keyword.

2. Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses I extensively in building its class library.

The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as *inheritance path.*

A derived class with multilevel base classes is declared as follows

```
{ .....
..... .
}
class B extends A // First level
{ .....
..... .
}
class C extends B // Second level
{ .....
..... .
```

}
This process may be extended to any number of levels. The class C can inherit the members of both A and B.

Hierarchical Inheritance
Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level. hierarchical classification of accounts in a commercial bank. This is possible because all the accounts posses certain common features.

5.(b) What are the 3 uses of keyword "final" ?Explain.

1. Using `final` to define constants: If you want to make a local variable, class variable (`static` field), or instance variable (non-static filed) constant, declare it `final`. A final variable may only be assigned to once and its value will not change and can help avoid programming errors.
Once a `final` variable has been assigned, it always contains the same value. If a `final` variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.
This also applies to arrays, because arrays are objects; if a `final` variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

2. Using `final` to prevent inheritance: If you find a class's definition is complete and you don't want it to be sub-classed, declare it `final`. A `final` class cannot be inherited, therefore, it will be a compile-time error if the name of a `final` class appears in the `extends` clause of another class declaration; this implies that a `final` class cannot have any subclasses.
It is a compile-time error if a class is declared both `final` and `abstract`, because the implementation of such a class could never be completed.
Because a `final` class never has any subclasses, the methods of a `final` class are never overridden

3.Using `final` to prevent overriding: When a class is extended by other classes, its methods can be overridden for reuse. There may be circumstances when you want to prevent a particular method from being overridden, in that case, declare that method `final`. Methods declared as `final` cannot be overridden.

5.(c) What are the uses of "this"?

Usage of java this keyword

1. this can be used to refer current class instance variable.

2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

6.(a) Briefly explain the thread life cycle with neat diagram. (2017-6A)

6.(b) Write a Java program to find area of any three geometrical figures using method overloading.

Example: Program to find area of Square, Rectangle and Circle using Method Overloading

```
class JavaExample
{
    void calculateArea(float x)
    {
        System.out.println("Area of the square: "+x*x+" sq units");
    }
    void calculateArea(float x, float y)
    {
        System.out.println("Area of the rectangle: "+x*y+" sq units");
    }
    void calculateArea(double r)
    {
        double area = 3.14*r*r;
        System.out.println("Area of the circle: "+area+" sq units");
    }
    public static void main(String args[]){
        JavaExample obj = new JavaExample();

        /* This statement will call the first area() method
         * because we are passing only one argument with
         * the "f" suffix. f is used to denote the float numbers
         *
         */
        obj.calculateArea(6.1f);

        /* This will call the second method because we are passing
         * two arguments and only second method has two arguments
         */
        obj.calculateArea(10,22);

        /* This will call the second method because we have not suffixed
         * the value with "f" when we do not suffix a float value with f
         * then it is considered as type double.
         */
        obj.calculateArea(6.1);
    }
}
```

Output:

```
Area of the square: 37.21 sq units
Area of the rectangle: 220.0 sq units
Area of the circle: 116.8394 sq units
```

7.(a) What is package? Explain how to create user defined package.

CREATING PACKAGES

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes.

We have seen in detail how Java system packages are organised and used. Now, let us see how to create our own packages. We must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file (except for comments and white spaces). Then we define a class, just as we normally define a
class. Here is an example:

```
package firstPackage;          // package declaration
public class FirstClass        // class definition
{ .....
..... (body of class)
..... ..
}
```

Here the package name is firstPackage. The class FirstClass is now considered a part of this package. This listing would be saved as a file called FirstClass.java, and located in a directory named firstPackage. When the source file is compiled, Java will create a .class file and store it in the same directory. Remember that the .class files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

To recap, creating our own package involves the following steps:

1. Declare the package at the beginning of a file using the form
2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates .class file in the subdirectory.

Remember that case is significant and therefore the subdirectory name must match the package name exactly. As pointed out earlier, Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots. Example:

```
package firstPackage.secondPackage;
```

This approach allows us to group related classes into a package and then group related

packages into a larger package. Remember to store this package in a subdirectory named FirstPackage \secondPackage.

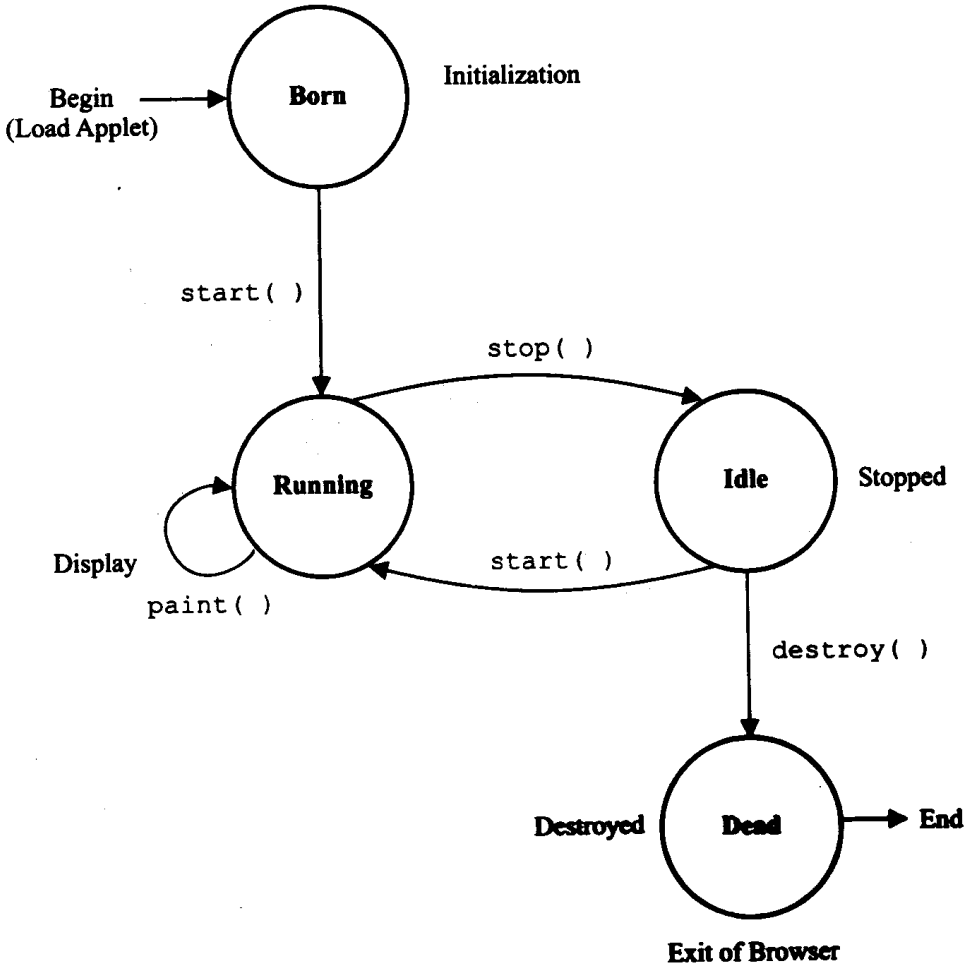
7.(b) Explain the applet life cycle with neat diagram.

APPLET LIFE CYCLE:

Every Java applet inherits a set of default behaviours from the **Applet** class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in Fig. 14.5. The applet states include:

- Born or initialization state
- Running state
- Idle state
- Dead or destroyed state

Fig. 14.5



An applet's state transition diagram

Initialization State

Applet enters the *initialization* state when it is first loaded. This is achieved by calling the init() method of Applet Class. The applet is born. At this stage, we may do the following, if required.

• Create objects needed by the applet
• Set up initial values
• Load images or fonts
• Set up colors

The initialization occurs only once in the applet's life cycle. To provide any of the behaviours mentioned above, we must override the init ( ) method:

```
public void init( )
{ .....
..... (Action)
..... .
}
```

Running State

Applet enters the *running* state when the system calls the start( ) method of Applet Class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in "stopped" (idle) state. For example, we may leave the Webpage containing the applet temporarily to another page and return back to the page. This again starts the applet running. Note that, unlike init( ) method, the start( ) method may be called more than once. We may override the start( ) method to create a thread to control the applet.

```
public void start ( )
{ .....
. . . . . (Action) .....
}
```

Idle or Stopped State

An applet becomes *idle* when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the stop( ) method explicitly. If we use a thread to run the applet, then we must use stop( ) method to terminate the thread. We can achieve this by overriding the stope) method.

```
public void stop( )
{ .....
..... (Action)
 .....
}
```

An applet is said to be *dead* when it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser. Like initialization, destroying stage occurs only once in the applet's life cycle. If the applet has created any resources, like threads, we may override the destroy( ) method to clean up these resources.

```
public void destroy( )
{ .....
..... (Action) ..... .
}
```

Display State

Applet .moves to the *display* state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The paint( ) method is called to accomplish this task. Almost every applet will have a paint( ) method.

Like other methods in the life cycle, the default version of paint( ) method does absolutely nothing. We must therefore override this method if we want anything to be displayed on the screen.

public void paint (Graphics g)
{ .....
..... (Display statements)
..... .
}

It is to be noted that the display state is not considered as a part of the applet's life cycle. In fact, the paint( ) method is not defined in the Applet class. It is inherited from the Component class, a super class of Applet.

8. Write Short notes about any four:

(a) Visibility controls

VISIBILITY CONTROL

We stated earlier that it is possible to inherit all the members of a class by a subclass using the keyword extends. We have also seen that the variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in Java by applying *visibility modifiers* to the instance variables and methods. The visibility modifiers are also known as *access modifiers.* Java provides three types of visibility modifiers: public, private and protected.

(b) Graphics Programming

One of the most important features of Java is its ability to draw graphics. We can write Java applets that draw lines, figures of different shapes, images, and text in different fonts and styles. We can also incorporate different colors in display. Every applet has its own area of the screen known as *canvas,* where it creates its display.

The size of an applet' s space is decided by the attributes of the <APPLET ... > tag. A Java applet draws graphical images inside its space using the coordinate system.

Java's Graphics class includes methods for drawing many different types of shapes, from simple lines to polygons to text in a variety of fonts. We have already seen how to display text using the paint( ) method and a Graphics object. To draw a shape on the screen, we may call one of the methods available in the Graphics class. All the drawing methods have arguments representing end points, corners, or starting locations of a shape as values in the applet's coordinate system. To draw a shape, we only need to use the appropriate method with the required arguments.

(c) Exception Handling

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental

errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block. This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, *Exception Type* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

(d) String methods

String Methods

The String class defines a number of methods that allow us to accomplish a variety of string manipulation tasks.

Some commonly used methods:

Method call	Task performed
s2 = sL.toLowerCase;	Converts the string sl. to all lowercase
52 = sl.toUpperCase;	Converts the .string sl to all Uppercase
s2 = sl. replace ('x' , 'y') ;	Replace all appearances of x with y
52 = 51. trim () ;	Remove white spaces at the beginning and end of the string sl
sl. equals (s2)	Returns 'true'if sl is equal to s2
sl.equalsIgnoreCase(s2)	Returns 'true'if sl = s2, ignoring the case of characters
sl . length ()	Gives the length of sl
51.ChartAt(n)	Gives nth character of sl
sl.compareTo(s2)	Returns negative if sl < s2, positive if sl > s2, and zero if sl is equal s2
sl.concat(s2)	Concatenates sl and s2
sl. substring (n)	Gives substring starting from nth character
Sl.5ubstring(n, m)	Gives substring starting from nth character up to mth

	(not including mth)
String.valueOf (p)	Creates a string object of the parameter p (simple type or object)
p. toString ()	Creates a string representation of the object p
sl.indexOf('x')	Gives the position of the first occurrence of 'x' in the string sl
sl.indexOf ('x' , n)	Gives the position of 'x' that occurs after nth position In the string sl
String.ValueOf(Variable)	Converts the parameter value to string representation.

(e) Interface

An interface is basically a kind of class. like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

Defining an 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. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list.

(F) Wrapper Classes

Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.