CBCS Scheme

USN □□□□□□□□□□

15CS561

Fifth Semester B.E. Degree Examination, Dec.2017/Jan.2018

Programming in Java

Time: 3 hrs. Max. Marks: 80

Note: Answer any FIVE full questions, choosing one full question from each module.

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

Module-1

1 a. Discuss three OOPs principles. **(06 Marks)**

b. Explain Java application Development step and JVM. **(06 Marks)**

c. Explain different Access Specifiers. **(04 Marks)**

OR

2 a. Explain the scope and life time of variables with an example **(05 Marks)**

b. What is narrowing and widening explain with an example. **(05 Marks)**

c. Explain how array in java work differently than C/C++. Write a java program to display

```
0   1   2   3   4
5   6   7   8   9
10  11  12  13  14
15  16  17  18  19
```

(06 Marks)

Module-2

3 a. Explain the following operator with an example.

i) Logical operator

ii) Bitwise operator **(05 Marks)**

b. With example explain about Ternary operator. Write a Java program to find the largest of three numbers using Ternary operator. **(06 Marks)**

c. Explain for each version of for loop write a program to display 2D student data: Name and USN. **(05 Marks)**

OR

4 a. Demonstrate the use of

* Continue statement in while loop

* Break statement in do while loop **(06 Marks)**

b. Write a java program to find the prime numbers form 1 to 100. **(05 Marks)**

c. Write a java program to perform simple calculator operation. **(05 Marks)**

Module-3

5 a. What are the salient features of constructor? Write a java program to show these features. **(05 Marks)**

b. Explain the following :

i) Use of this keyword

ii) Garbage collection in java

iii) Finalize ( ) method. **(06 Marks)**

c. With a java program show how final keyword is used to prevent inheritance and overriding. **(05 Marks)**

OR

6 a. Create a java class called Student with the following details as variables (USN, Name, Branch, Phone number). Write a java program to create n student object and print USN, Name Branch and phone number with suitable message. **(06 Marks)**

b. Differentiate between C++ and Java with respect to inheritance. Mention the use of supper class and this parameter in java with example. **(05 Marks)**

c. What is Inner class? Demonstrate with an example. **(05 Marks)**

Module-4

7 a. Define Interface. Explain how to define implement and assign variable in interface to perform "one interface multiple method" . **(05 Marks)**

b. What is the role of interface while implementing multiple inheritances in java? **(05 Marks)**

c. Write short note on :

i) Importing package

ii) Accesses protection. **(06 Marks)**

OR

8 a. Define Exception Demonstrate the working of nested try block with an example. **(06 Marks)**

b. Write a program which contains one method which will throw IllegalAccessException and use proper exception handles so that exception should be printed. **(05 Marks)**

c. Explain the following :

i) Java's built in Exception

ii) Uncaught Exception **(05 Marks)**

Module-5

9 a. What are Applets? Explain the different stages in the life cycle of Applet. **(06 Marks)**

b. Write a note :

i) Type Wrapper

ii) Transient and volatile modifier **(05 Marks)**

c. Enlist the Applet Tag. **(05 Marks)**

OR

10 Explain the following with example

a. String co

b. mparision

c. Searching strings

d. Modifying string

Overloading constructor. **(16 Marks)**

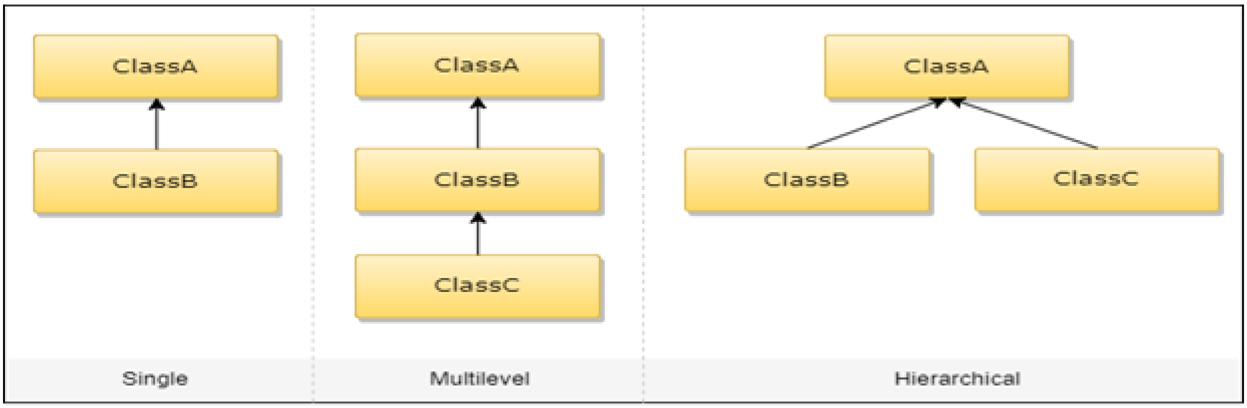
* * * * *

1) A) Discuss three OOP principles. [2+2+2]

Encapsulation: Encapsulation can be defined as the procedure of casing up of codes and their associated data jointly into one single component.
In simple terms, encapsulation is a way of packaging data and methods together into one unit. Encapsulation gives us the ability to make variables of a class keep hidden from all other classes of that program or namespace.
Hence, this concept provides programmers to achieve data hiding. Programmers can have full control over what data storage and manipulation within the class

Inheritance: Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

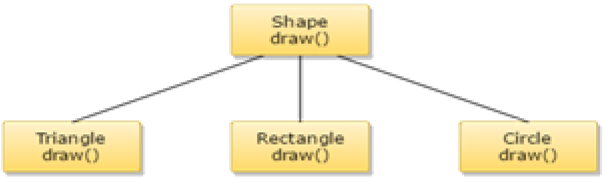
Java supports three types of inheritance. These are:



Java Inheritance

Polymorphism: The word polymorphism means having multiple forms. The term Polymorphism gets derived from the Greek word where poly + morphos where poly means many and morphos means forms.

- Static Polymorphism



Polymorphism

- Dynamic Polymorphism.

b) Explain steps to execute simple java program and role of JVM in execution

Steps:

a) After typing the program in the terminal we have to type javac progranmane.java and hit enter

b) Then again in the terminal (if no error is there) the we have to type java program name and hit enter key

Role of JVM in execution can be described as follows:

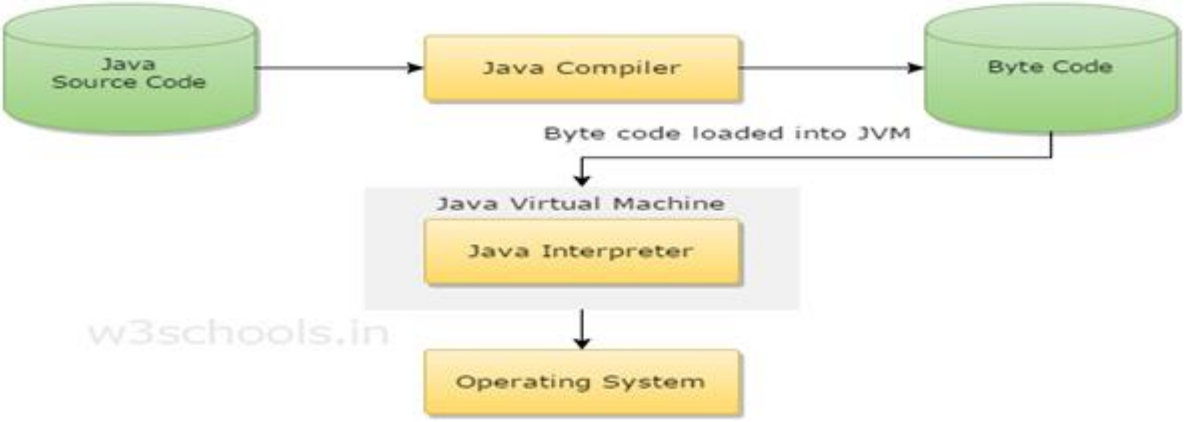


Diagram of JVM

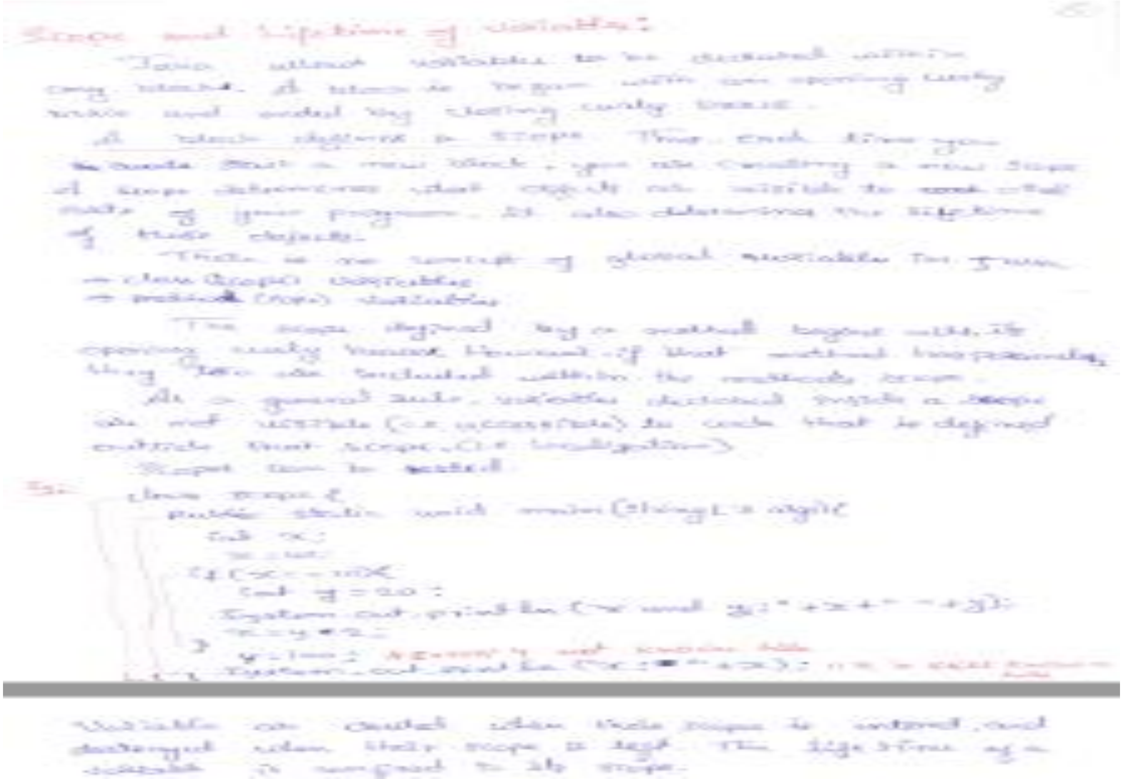
- Reading Bytecode.
- Verifying bytecode.

Linking the code with the library

c) Explain different access specifiers

- **Public: keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.**
- **Default: (No visibility modifier is specified): it behaves like public in its package and private in other packages.**
- **Default Public keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.**
- **Private : fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.**
- **Protected : members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.**

2) A) Explain the scope and the life time of a variable with an example [05]



Scope and lifetime of variables:

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by closing curly brace.

A block defines a scope. Thus each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

There is no concept of global variables in Java.
→ class scope variables
→ method (local) variables

The scope defined by a method begins with its opening curly brace. However if that method has parameters, they too are included within the method's scope.

As a general rule, variables declared inside a scope are not visible (i.e. accessible) to code that is defined outside that scope (i.e. localization). Scopes can be nested.

Ex:

```
class Scope {
  public static void main(String[] args){
    int x;
    x = 10;
    if(x == 10){
      int y = 20;
      System.out.println("x and y: "+x+" "+y);
      x = y*2;
    }
    y = 100; // Error! y not known here
    System.out.println("x is "+x);
```

Variables are created when their scope is entered, and destroyed when their scope is left. The lifetime of a variable is confined to its scope.

b) What is narrowing and widening. Explain with an example.
[6]

Widening : When one type of data is assigned to another type of variable, an automatic type conversion

will take place if the following two conditions are met:
• The two types are compatible.
• The destination type is larger than the source type.
When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.

Narrowing: In case of stornt int value to a byte variable, conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, To create a conversion between two incompatible types, we must use a cast. A cast is simply an explicit type conversion. It has this general form:
(target-type) value
For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// ...
b = (byte) a;
```

c) Explain how array in java works differently than C(syntax and examples are compulsory). Write java program to display [5]

```
 0  1 2 3 4
 5 6 7 8  9
10 11 12 13 14
15 16 17 18 19
```

[6]

In C programming declaration of array is as follows:

```
Datatype arr-name[size of the array];
int  arr[20];
```

But in array is declared as follows:

```
Datatype arr-name[ ]= new Datatype [size of the array];
Int arr[ ]= new arr[20];
```

In java array will be declared with the keyword "new" and for that all the elements in the array will be initialized to 0. But in C as no new keyword is used at the time of array declaration so elements of array in C at the time of declaration will be initialized to null.

java program to display

```
 0  1 2 3 4
 5 6 7 8  9
10 11 12 13 14
15 16 17 18 19
```

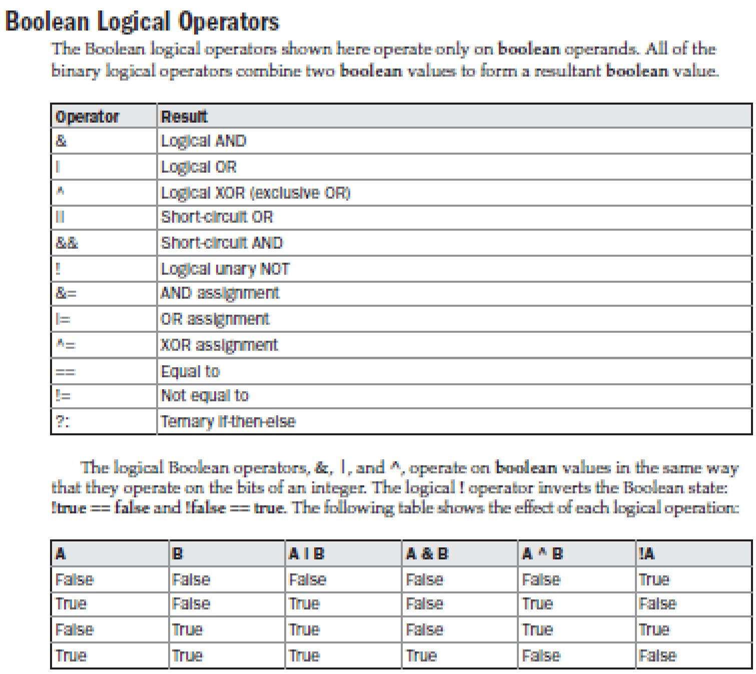
```
public class display {

    public static void main(String[] args) {
    for (int i=0;i<20;i++)
    {
        System.out.print(i+" ");
        if(i==4)
        {
        System.out.println();
        }
    }
    }

}
```

3) A) **Explain the following operators with example**

- **Logical opeartor**
- **Bitwise operator**



Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Bitwise opeator

~	**~op**	**Inverts all bits**
&	**op1 & op2**	**Produces 1 bit if both operands are 1**
|	**op1 |op2**	**Produces 1 bit if either operand is 1**
^	**op1 ^ op2**	**Produces 1 bit if exactly one operand is 1**
>>	**op1 >> op2**	**Shifts all bits in op1 right by the value of op2**
<<	**op1 << op2**	**Shifts all bits in op1 left by the value of op2**

b) Explain Ternary operator. Write a java program to display largest of three numbers using ternary operator [6]

- ternary (three-way) operator that can replace certain types of if-then-else statements.
- This operator is the ?.
- The ? has this general form: expression1 ? expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.
- The result of the ? operation is that of the expression evaluated.
- Both expression2 and expression3 are required to return the same type, which can’t be void.

Largest among three numbers:

```
public class Ternary {

        public static void main(String[] args) {
        int a=40,b=39,c=99,res;

        res=(a>b)?((a>c)?a:c):((b>c)?b:c);
        System.out.println(res);
        }

}
```

c) Explain use of for..each loop with suitable example. Write a program to explain a 2D data: Name and USN

Syntax

Following is the syntax of enhanced for loop −

```
for(declaration : expression) {
   // Statements
}
```

- Declaration − The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- Expression − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

```
import java.util.Scanner;

public class GetStudentDetails
{
      public static void main(String args[])
      {
            String name;
            int roll, math, phy, eng;

            Scanner SC=new Scanner(System.in);

            System.out.print("Enter Name: ");
            name=SC.nextLine();
            System.out.print("Enter Roll Number: ");
            roll=SC.nextInt();
            System.out.print("Enter marks in Maths,
Physics and English: ");
            math=SC.nextInt();
            phy=SC.nextInt();
            eng=SC.nextInt();
```

```
          int total=math+eng+phy;
          float perc=(float)total/300*100;

          System.out.println("Roll Number:" + roll
+"\tName: "+name);
          System.out.println("Marks (Maths, Physics,
English): " +math+","+phy+","+eng);
          System.out.println("Total: "+total
+"\tPercentage: "+perc);

      }

}
```

6) a) **Demonstrate use of i) break statement in do..while loop ii) continue statement in while loop with examples.**

The break statement can be used with any of Java’s loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a while loop. The output from this program is the same as just shown.

```
// Using break to exit a while loop.
class BreakLoop2 {
public static void main(String args[]) {
int i = 0;
do{
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
i++;
}while(i < 100);
System.out.println("Loop complete.");
}
}
```

In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop

For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
public static void main(String args[]) {
Int i=0;
while( i<10)
{
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");

i++ ;
}
}
}
```

b) **Write program to display Prime numbers between 1 to 100.**

```
class PrimeNumbers
{
   public static void main (String[] args)
   {
       int i =0;
       int num =0;
       //Empty String
       String  primeNumbers = "";

       for (i = 1; i <= 100; i++)
       {
          int counter=0;
          for(num =i; num>=1; num--)
        {
             if(i%num==0)
           {
            counter = counter + 1;
           }
        }
        if (counter ==2)
        {
           //Appended the Prime number to the String
           primeNumbers = primeNumbers + i + " ";
        }
       }
       System.out.println("Prime numbers from 1 to 100 are :");
       System.out.println(primeNumbers);
   }
}
```

c) Simple calculator

```
import java.util.Scanner;

public class JavaExample {

    public static void main(String[] args) {

        double num1, num2;
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number:");

        /* We are using data type double so that user
         * can enter integer as well as floating point
         * value
         */
        num1 = scanner.nextDouble();
        System.out.print("Enter second number:");
        num2 = scanner.nextDouble();

        System.out.print("Enter an operator (+, -, *, /): ");
        char operator = scanner.next().charAt(0);

        scanner.close();
        double output;

        switch(operator)
        {
            case '+':
                 output = num1 + num2;
                break;

            case '-':
                 output = num1 - num2;
                break;

            case '*':
                 output = num1 * num2;
                break;

            case '/':
                 output = num1 / num2;
                break;

            /* If user enters any other operator or char apart from
             * +, -, * and /, then display an error message to user
             *
             */
            default:
                System.out.printf("You have entered wrong operator");
                return;
        }

        System.out.println(num1+" "+operator+" "+num2+": "+output);
    }
}
```

Module-3

5)a)

Discuss the salient features and types of Constructor with programming example. **[1**

- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor
- A constructor initializes the instance variables of an object.
- It is called automatically immediately after the object is created but before the new operator completes.
  1) it is syntactically similar to a method:
  2) it has the same name as the name of its class
  3) it is written without return type, not even void; the default return type of a class

In Box example the dimensions of a box are automatically initialized when an object is constructed.

```
class Box {
    double width;
    double height;
    double depth;
    Box() {
            System.out.println("Constructing Box");
            width = 10; height = 10; depth = 10;
            }
            double volume() {
                    return width * height * depth;
                    }
            }
  public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            vol = mybox1.volume();
            System.out.println("Volume is " + vol);
            vol = mybox2.volume();
            System.out.println("Volume is " + vol);
            }
    }
```

it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

constructor for the class is being called

Parameterized Constructor: The constructor which takes parameter while creating the object of a particular class. Here Box constructor is having width, height and depth parameters which will be initialized at the time of object creation which the supplied values.

```
class Box {
```

```
            double width;
            double height;
            double depth;
            Box(double w, double h, double d) {
                        width = w; height = h; depth = d;
                                    }
            double volume(){
                         return width * height * depth;
                                    }
                        }
  class BoxDemo {
            public static void main(String args[]) {
                        Box mybox1 = new Box(10, 20, 15);
                        Box mybox2 = new Box(3, 6, 9);
                        double vol;
                        vol = mybox1.volume();
                        System.out.println("Volume is " + vol);
                        vol = mybox2.volume();
                        System.out.println("Volume is " + vol);
}
}
```

```
Volume is 3000.0
Volume is 162.0
```

b)

Explain the following:
a) Use of this keyword
b) Garbage collector
c) Finalize ()

This Keyword:

- Sometimes a method will need to refer to the object that invoked it
- this is always a reference to the object on which the method was invoked
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors
  - Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
            Box(double w, double h, double d) {
                        this.width = w;
                        this.height = h;
                        this.depth = d;
                        }
```

From the main function we can construct the Box object by:

```
public static void main(String args[]) {
                        Box mybox1 = new Box(10,20,30);
                        Box mybox2 = new Box(myBox1);  //here the myBox1 object has been passed
```

This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. section. This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. when a local variable has the same name as an instance variable, the local variable hides the instance variable.

Garbage Collection:

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:

1) The garbage collector keeps track of how many references an object has.
2) when no references to an object exist, that object is assumed to be no longer needed, and the memory
occupied by the object can be reclaimed.

3) It removes an object from memory when it has no longer any references.
4) Thereafter, the memory occupied by the object can be allocated again.
5) The garbage collector invokes the finalize method.

Finalize() :

- .
- Sometimes an object will need to perform some action when it is destroyed. To handle such situations, Java provides a mechanism called finalization
- the finalize method is invoked just before the object is destroyed:
- By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, we simply define the finalize( ) method. The
- Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.
- implemented inside a class as:
- protected void finalize() { … }
- implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out.
- Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class
- It is important to understand that finalize( ) is only called just prior to garbage collection.

c)

1) **Using final to Prevent Overriding::::**

- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden. The following fragment illustrates final:

```
class A {
   final void meth() {
      System.out.println("This is a final method.");
                }
 }
   class B extends A {
        void meth() { // ERROR! Can't override.
             System.out.println("Illegal! ");
                          }
             }
```

- Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
- Inlining is only an option with final methods.
- Normally, Java resolves calls to methods *dynamically, at run time*. This is called ***late binding***. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called ***early binding.***

- **Using final to Prevent Inheritance::::**
- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
-
- Declaring a class as final implicitly declares all of its methods as final, too.
- As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Here is an example of a final class:

```
      final class A {
           // ...
         }
// The following class is illegal.
         class B extends A { // ERROR! Can't subclass A
              // ...
           }
```

- As the comments imply, it is illegal for B to inherit A since A is declared as final.

6)a)

```
 import java.io.*;
class Student
   {
      int rollno;
      String name;
      int number_of_subjects;
      int mark[];
     Student(int roll,String stud_name,int noofsub) throws IOException
        {
           rollno=roll;
           name=stud_name;
           number_of_subjects= noofsub;
           getMarks(noofsub);
        }
    public void getMarks(int nosub ) throws IOException
      {
          mark=new int[nosub];
          BufferedReader br= new BufferedReader (new InputStreamReader(System.in));
          for (int i=0; i<nosub;i++)
              {
               System.out.println(“Enter “+i+”Subject Marks.:=> “);
                 mark[i]=Integer.parseInt(br.readLine());
               System.out.println(“”);
              }
        }
  public void calculateMarks()
    {
      double percentage=0;
      String grade;
      int tmarks=0;
      for (int i=0;i<mark.length;i++)  {
         tmarks+=mark[i];
      }
    percentage=tmarks/number_of_subjects;
    System.out.println(“Roll Number :=> “+rollno);
    System.out.println(“Name Of Student is :=> “+name);
    System.out.println(“Number Of Subject :=> “+number_of_subjects);
    System.out.println(“Percentage Is :=> “+percentage);
    if (percentage>=70)
      System.out.println(“Grade Is First Class With Distinction “);
    else if(percentage>=60 && percentage<70)
      System.out.println(“Grade Is First Class”);
    else if(percentage>=50 && percentage<60)
     System.out.println(“Grade Is Second Class”);
     else if(percentage>=40 && percentage<50)
      System.out.println(“Grade Is Pass Class”);
    else
     System.out.println(“You Are Fail”);
       }
   }

class StudentDemo  {
     public static void main(String args[])throws IOException {
          int rno,no,nostud;
          String name;
          BufferedReader br= new BufferedReader (new InputStreamReader(System.in));
         System.out.println(“Enter How many Students:=> “);
          nostud=Integer.parseInt(br.readLine());
         Student s[]=new Student[nostud];
         for(int i=0;i<nostud;i++)   {
           System.out.println(“Enter Roll Number:=> “);
            rno=Integer.parseInt(br.readLine());
           System.out.println(“Enter Name:=> “);
            name=br.readLine();
           System.out.println(“Enter No of Subject:=> “);
no=Integer.parseInt(br.readLine());
s[i]=new Student(rno,name,no);
                }
for(int i=0;i<nostud;i++)  {
      s[i].calculateMarks();
                          }
                }
        }
```

b)
4 **What is Inheritance? What are the different types of Inheritance? Explain the use of super with a suitable programming example.** **[5+**

Inheritance:

- Allows the creation of hierarchical classifications.
- For creating inheritance:

inheritance is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes.

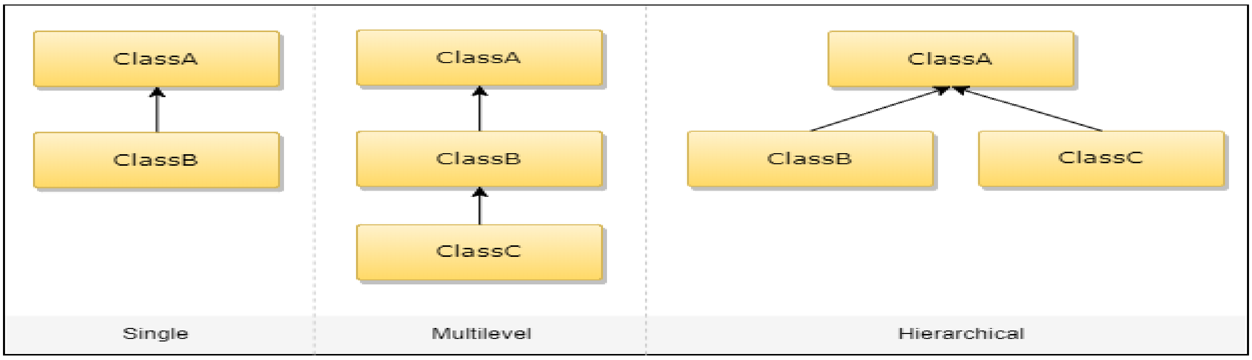
A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass. A subclass can have only one superclass, whereas a superclass may have one or more subclasses.

- A class that is inherited is called a superclass. The class that does the inheriting is called a subclass.
- Subclass is a specialized version of a superclass.
- To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword
- The general form of a class declaration that inherits a superclass is shown here:

```
Class subclass-name extends superclass-name {
        // body of class
    }
```

A is a superclass for B (In next slide eg.), it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself.

- Further, a subclass can be a superclass for another subclass.



Java Inheritance

Super:

- super is a keyword.
- It is used inside a sub-class method definition to call a method defined in the super class. Private methods of the super-class cannot be called. Only public and protected methods can be called by the super keyword.
- It is also used by class constructors to invoke constructors of its parent class.
- Super keyword are not used in static Method.

Two general form of super:

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass

- A subclass can call a constructor defined by its superclass by use of the following form of super:

```
            super(arg-list);     //arg-list specifies any arguments needed by the constructor in the superclass.
```

- super( ) must always be the first statement executed inside a subclass' constructor
- Example:

a) First use of super : to call super class constructor:

```
class Box{
  double height;
  double widht;
  double depth;
 Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;

  double vol()
      {
      // body of the function
    }
}
```

```
class BoxWeight entends Box{
    double mass;
    BoxWeight(double w, double h, double d, double m) {
            super(w,h,d);
         mass=m ;
      }
 }
```

Here the value of height,width, depth in Boxweight contructor has been initialized by super class constructor with "super".

b) Second use of super : to call super class constructor:

- This second form of super is most applicable to situations when member names of a subclass hide members by the same name in the superclass
- This usage has the following general form:

```
          super.member                    //Here, member can be either a method or an instance variable.
```

- **This :** Sometimes a method will need to refer to the object that invoked it
- this is always a reference to the object on which the method was invoked
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors
  - Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
        Box(double w, double h, double d) {
                this.width = w;
                this.height = h;
                this.depth = d;
                }
```

Use this to resolve name-space collisions.

```
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
        }
```

when a local variable has the same name as an instance variable, the local variable hides the instance variable

c) Inner class with an example

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Nested Inner class
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
Like class, interface can also be nested and can have access specifiers.
Following example demonstrates a nested class.

```
class Outer {
   // Simple nested inner class
   class Inner {
      public void show() {
           System.out.println("In a nested class method");
      }
   }
}
class Main {
   public static void main(String[] args) {
       Outer.Inner in = new Outer().new Inner();
       in.show();
   }
}
```

Module-4

7)a) **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”.

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
```

c) **Access Protection**

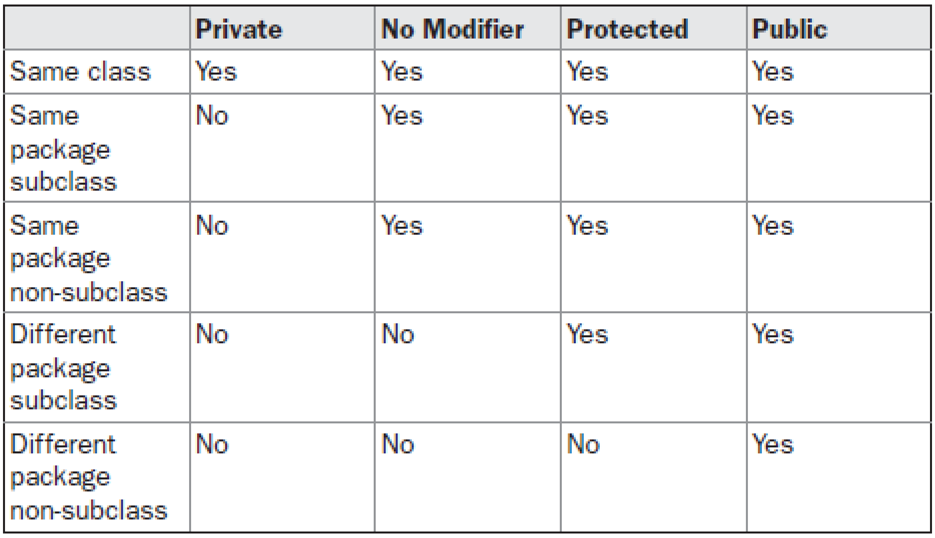
In the preceding chapters, you learned about various aspects of Java’s access control mechanism and its access specifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. 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. Table 9-1 sums up the interactions. While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

TABLE 9-1 Class Member Access



	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

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

import *pkg1*[.*pkg2*].(*classname*|*);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

8) a)

Exception is an abnormal condition that arises in the code sequence.

- Exceptions occur during compile time or run time.
- “throwable” is the super class in exception hierarchy.
- Compile time errors occurs due to incorrect syntax.
- Run-time errors happen when
  - User enters incorrect input
  - Resource is not available (ex. file)
  - Logic error (bug) that was not fixed

Nested try

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

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.

If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
```

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

b)

```
        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
```

c)

i) Built-in Exception

- Inside the standard package **java.lang, Java defines several exception classes**
- The most general of these exceptions are subclasses

of the standard type **RuntimeException.** These exceptions need not be included in any method's **throws list. In the language of Java, these are called** *unchecked exceptions because the compiler does not check to see if a method handles or* throws these exceptions.

- *checked exceptions* those exceptions defined by **java.lang that must be included** in a method's **throws list if that method can generate one of these exceptions and does** not handle it itself.

ii) Uncaught Exception

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

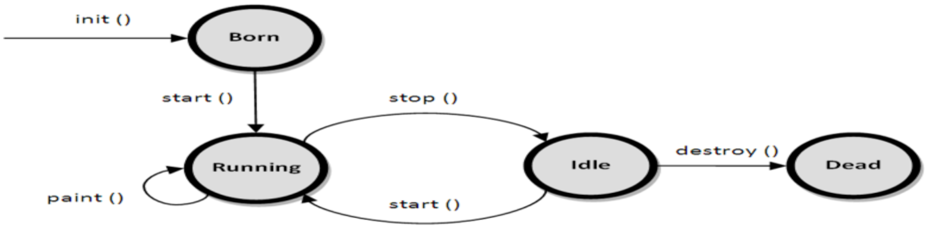
When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Module-5

9)a)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()

b)Type wrappers

Using objects primitive data type values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit Object.

- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.
- To handle these (and other) situations, Java provides ***type wrappers***, which are classes that encapsulate a primitive type within an object.
- The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character, and Boolean.** These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Transient and volatile modifiers

Java defines two interesting type modifiers: transient and volatile.
These modifiers are used to handle somewhat specialized situations.
***Transient Modifiers*:** When an instance variable is declared as transient, then its value need not persist when an object is stored. For example:

```
class T { transient int a; // will not persist
 int b; // will persist }
```

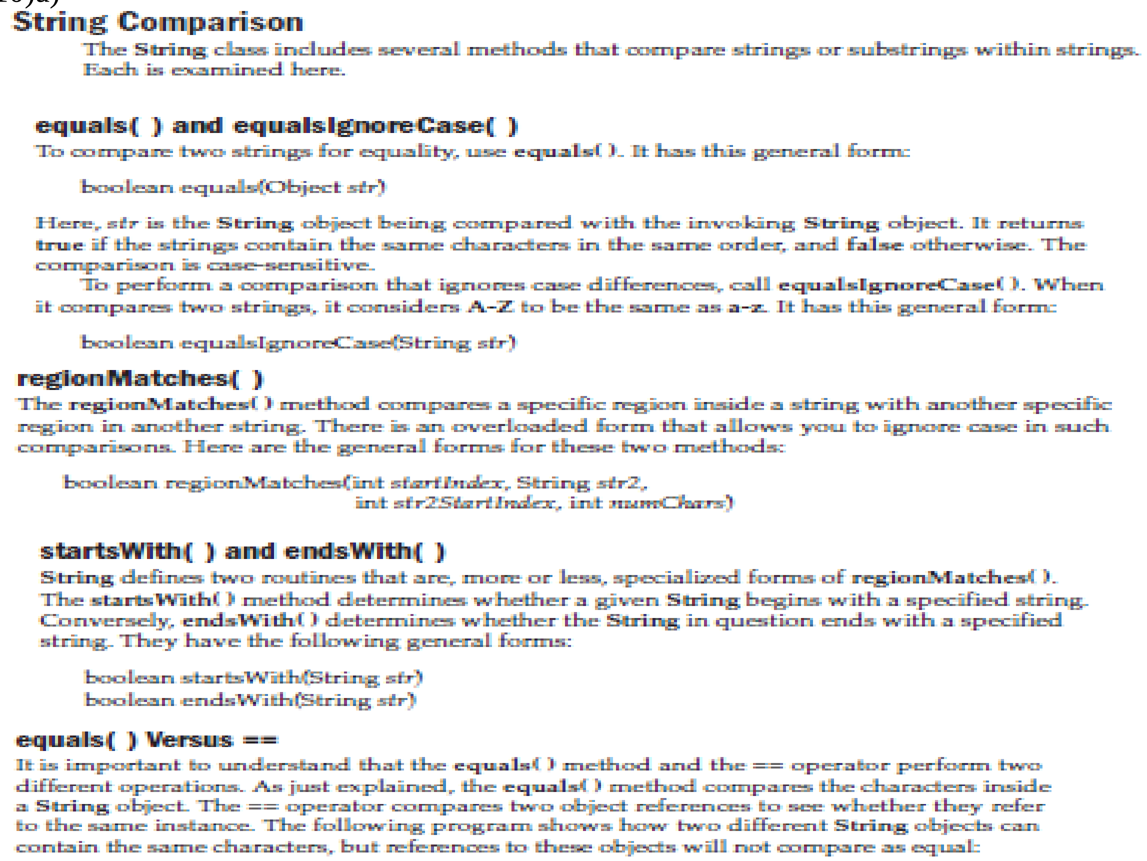
Here, if an object of type T is written to a persistent storage area, the contents of a would not be saved, but the contents of b would.

Volatile modifier: The volatile modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of the program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable.

For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered.
All that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as volatile, which tells the compiler that it must always use the master copy of a volatile variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy

c) Applet tags
Code
Width Height
Name
Codebase
Alt
Param
Vspace
Hspace
Align

10)a)



String Comparison

The **String** class includes several methods that compare strings or substrings within strings. Each is examined here.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2,
                      int str2StartIndex, int numChars)
```

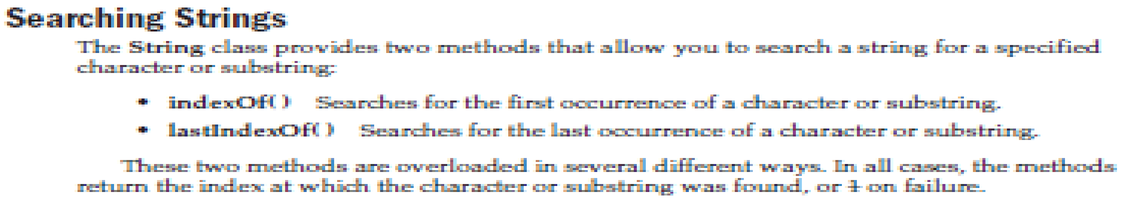
startsWith( ) and endsWith( )

String defines two routines that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)
boolean endsWith(String *str*)

equals( ) Versus ==

It is important to understand that the **equals( )** method and the == operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:



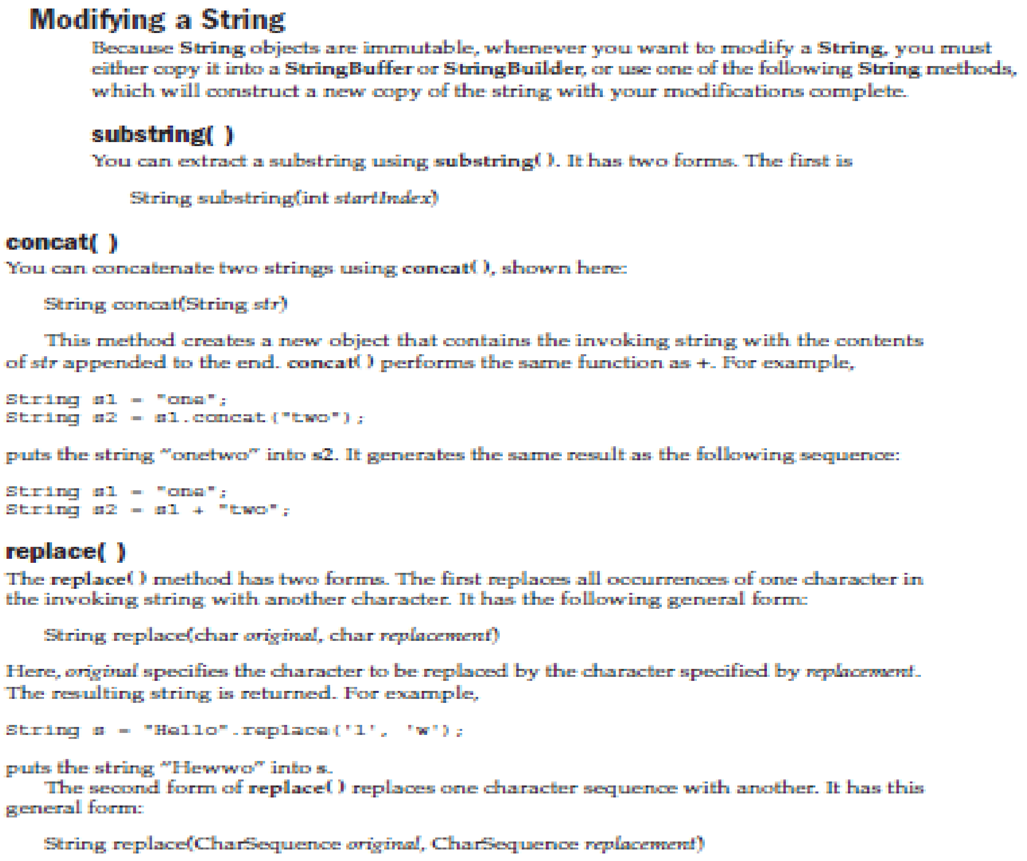
Searching Strings

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.

b)



Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with your modifications complete.

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into s.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

Overloading contractor

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the ***this*** keyword.

The general form is shown here:

this(arg-list)

When ***this( )*** is executed, the overloaded constructor that matches the parameter list specified by arg-list is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to ***this( )*** must be the first statement within the constructor. Take the following example which has two versions: