Q2 (a) List four major differences between C++ and Java.

Answer Page Number 15 of 4th edition of textbook.

Q2 (b) what are the following components used for?

(i) javac	**(ii) java**
(iii) appletviewer	**(iv) jdb**

Answer Page Number 28 of 4th edition of textbook.

Q3 (a) Explain iteration statements in java with suitable examples.

Answer
Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met

while

The **while** loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
  // body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.
Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println( )** is never executed:

```
int a = 10, b = 20;

while(a > b)
  System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java's loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```
// The target of a loop can be empty.
class NoBody {
```

© IETE

```
    public static void main(String args[]) {
      int i, j;

      i = 100; j = 200;

      // find midpoint between i and j
      while(++i < --j) ; // no body in this loop

      System.out.println("Midpoint is " + i);
    }
  }
```

This program finds the midpoint between **i** and **j**. It generates the following output:

```
Midpoint is 150
```

Here is how the **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops.Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
  // body of loop } while
(condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression. Here is a reworked version of the "tick" program that demonstrates the **do-while** loop. It generates the same output as before.

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do {
        System.out.println("tick " + n);
} while(--n > 0);
```

In this example, the expression **(- -n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **- -**n statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

For:

the **for** statement:

```
for(initialization; condition; iteration) {
 // body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false. When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the

for loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop. When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed elsewhere.

```
// Test for primes.
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime = true;

        num = 14;
        for(int i=2; i <= num/2; i++) {
            if((num % i) == 0) {
                isPrime = false;
                break;
            }
        }
        if(isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
    }
}
```

Q3 (b) Explain break and continue statements in detail with examples.

Answer

In Java, the **break** statement has three uses. First, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

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

```
        System.out.println("Loop complete.");
    }
}
```

The **break** statement can be used with any of Java's loops, including intentionally infinite loops.
Two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.
When used inside a set of nested loops, the break statement will only break out of the innermost loop. For example:

```
// Using break with nested loops.
class BreakLoop3
{
  public static void main(String args[]) {
   for(int i=0; i<3; i++)
     {
       System.out.print("Pass " + i + ": ");
         for(int j=0; j<100; j++)
           {
             if(j == 10) break; // terminate loop if j is 10
             System.out.print(j + " ");
           }
       System.out.println();
     }
   System.out.println("Loops complete.");
  }
}
```

Using break as a Form of Goto :

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement. Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner Java defines an expanded form of the **break** statement. By using this form of **break**, you can break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

break *label*;

Here, *label* is the name of a label that identifies a block of code. When this form of

break executes, control is transferred out of the named block of code. The labeled block of code must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control to a block of code that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println( )** statements.

```
// Using break as a civilized form of goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

Using continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

© IETE

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[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0
  1
2
  3
4
  5
6
  7
8
  9
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue.

Q4 (a) Design a class to represent an employee. Include following data members and methods:

Variables:
Name of employee
Age
Sex
DOB(dd/MM/yyyy)
Basic Salary
Marital status
Constructor:
to assign initial values
Methods:to display various details about employee

© IETE

Answer

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Employee {

private String strEmployeeName = "";
private String strSex;
private String strMaritalStatus;
private String[] strFamilyMemberNames;
private Date DOB;
private float floatAge;
private float floatBasicSalary ;

private static final int PF = 50;
private static final int HRA = 25;
private static final int TA = 15;
private static final float TAX = 17.5f;

public Employee(String strEmployeeName, String strSex, String
strMaritalStatus, String[] strFamilyMemberNames, Date DOB, float floatAge,
float floatBasicSalary)
{
this.strEmployeeName = strEmployeeName;
this.strSex = strSex;
this.strMaritalStatus = strMaritalStatus;
this.strFamilyMemberNames = strFamilyMemberNames;
this.DOB = DOB;
this.floatAge = floatAge;
this.floatBasicSalary = floatBasicSalary;
}

public float getInHandSalary()
{
float inHandSalary = 0.0f;
inHandSalary = floatBasicSalary - (floatBasicSalary*(PF/100)) -
(floatBasicSalary*(TAX/100))
+ (floatBasicSalary*(HRA/100)) + (floatBasicSalary*(TA/100));
return inHandSalary ;
}

@SuppressWarnings("finally")
public int getAccurateAge()
```

```
{
int accurateAge = 0;
int factor = 0;
try
{

SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
sdf.format(new Date());
Date currentDate = sdf.parse(sdf.format(new Date()));
Calendar currentDateCal = Calendar.getInstance();
Calendar dobCal = Calendar.getInstance();
currentDateCal.setTime(currentDate);
dobCal.setTime(DOB);

if(currentDateCal.get(Calendar.DAY_OF_YEAR) <
dobCal.get(Calendar.DAY_OF_YEAR)) {
factor = -1;
}
accurateAge = currentDateCal.get(Calendar.YEAR) -
dobCal.get(Calendar.YEAR) + factor;

return accurateAge;
}
catch (ParseException pe)
{
// TODO Auto-generated catch block
pe.printStackTrace();
}
catch (Exception e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}
finally
{
return accurateAge;
}
}

public void showEmployeeDetail()
{
System.out.println("Employee Name : "+strEmployeeName);
System.out.println("Sex : "+strSex);
System.out.println("Marital Status : "+strMaritalStatus);
System.out.println("DOB : "+DOB);
```

© IETE

```
System.out.println("Age : "+floatAge);
System.out.println("Basic Salary : "+floatBasicSalary);
System.out.println("In Hand Salary : "+getInHandSalary());
System.out.println("Name of the fanily members are as follow : ");
for(int loop = 0 ; loop < strFamilyMemberNames.length ; loop++)
{
System.out.println((loop+1)+". "+strFamilyMemberNames[loop]);
}
}
}
```

Q4 (b) what is inheritance and purpose of using inheritance? What is the use of keyword super in inheritance? Explain with an example.

Answer
Interfaces:
An interface in java can be considered as a contract or way of saying what a class can do without saying anything about how the class will do it.

Interfaces are defined using keyword "interface" in java.
Syntax for declaring syntax is as follow:

```
public interface Test
{
        public static final int PI = 3.343;
        public abstract void doSomething();
}
```

Following are the key points about interfaces:

1. Like classes, interfaces can have public or default access.
2. All the variables defined in the interfaces are public static final by defaul, we don't need to use these keywords explicitly, in fact using these keywords are considered redundant. As we know in Java final keyword is used to declare constants, so interface can have only constants not the instance variables.
3. An interface can contain only method declaration not definition. All the methods defined inside are public and abstract by default, we don't need to explicitly mention these. Using these keywords are considered redundant.

So following interface declaration is exactly same as the previous one:

```
public interface Test
{
        int PI = 3.343;
        void doSomething();
```

© IETE

```
}
```

4. An interface can extend another interface as follow:

```
public  interface TestChild extends Test
{
        int  X = 15;
}
```

5. A class can implement any number of interfaces as follow:

```
public class TestInterface implements Test, Serializable
{
        //class code goes here
}
```

Any class that implements an interface must provide implementations of all the methods declared inside interface. The only exception is for abstract class. Even in that case the first concrete class has to provide implementation of all abstract methods above itself in the inheritance tree.

6. Interface can not be declared as static and can't have constructors because they are not classes.

Advantage of using Interface over Abstract classes:

An abstract class is a class which can not be instantiated. It may or may not have abstract methods. An abstract method is one which is being declared not defined using keyword "abstract" and any class extending abstract class must provide implementation of abstract methods if any.

And on the other hand an interface can be considered as 100% abstract class.
The only advantage of using Interface over abstract class is that as we know java does not support multiple inheritance, means a class can only have one parent. So interfaces are useful in that situation, means if our class is already extending another class say Thread and we need some functionality from other class in that case we can declare interfaces and make class to implement that interface.

Q5 (a) What is an interface in java? Differentiate between interface and abstract classes.

Answer
Inheritance and its purpose:
Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As

mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog,* which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.
Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization.

This is a key concept which lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.
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
}
You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. (This differs from C++, in which you can inherit multiple base classes.) You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Using super in inheritance:
There will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
super has two general forms. 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.

Using super to Call Superclass Constructors

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

super(*parameter-list*);

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight uses super to initialize its Box attributes.
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

The key concepts behind **super( )** is when a subclass calls **super( )**, it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
E.g

```
// Using super to overcome name hiding.
class A {
int i;
}

// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A

B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
```

```
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}

class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);

subOb.show();
}
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Q5 (b) What is difference between multithreading and multitasking? Explain life cycle of a thread in java.

Answer

Multitasking and Multithreading In java:

Multitasking:

1. A process can be defined as a program under execution. Multitasking can be defined as executing two or more programs together, simultaneously. For eg. In our system we are running browser, medial player and MSOffice applications simultaneously.
2. A process is much more high weight then a thread.
3. Each process requires its one resources from system

Multithreading:

1. A thread can be defined as an independent chunk of code written to perform some tasks. Executing multiple threads simultaneously is known as multithreading. Java supports multithreading. For eg. Threads are mostly used in gaming applications, while one frame is executing, some other task may be preparing next frame.
2. A thread is much lighter than a process.

Each thread share resources provided to the program of which they are a part.

© IETE

Q6 (a) What is an exception? Explain with suitable example how a program throws an exception explicitly?

Answer

An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.
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.**

It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.
Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
```

```
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects.

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

Q6 (b) Write a java program to determine whether a string is palindrome or not.

Answer

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Palimdrome {
private BufferedReader br ;
public Palimdrome()
 {
        br = new BufferedReader(new InputStreamReader(System.in));
 }

 public String readData() throws IOException
 {
        String enteredString = "";
```

```
        System.out.println("Please enter a string....");
        enteredString = br.readLine();
        System.out.println("Thanks for your Iputs!!!");
        System.out.println("String entered by you is : "+enteredString);
        return enteredString ;
}

public void checkPalindrome(String enteredString)
{
        char[] enteredStringArray = new char[enteredString.length()];
        if(enteredString.length() == 1)
        {
                System.out.println("String is Palimdrome as it contains only single
character.");
        }
        else
        {
                for(int loop = 0 ; loop < enteredString.length() ; loop++)
                {
                        enteredStringArray[loop] = enteredString.charAt(loop);
                }
                for(int loopStart = 0, loopEnd = enteredStringArray.length-1 ; loopStart <
enteredStringArray.length ; loopStart++,loopEnd--)
                {
                        if(loopStart <= loopEnd)
                        {
                                if(enteredStringArray[loopStart] ==
enteredStringArray[loopEnd])
                                {
                                        continue;
                                }
                                else
                                {
                                        System.out.println("String is not palindrome");
                                        break;
                                }
                        }
                        else
                        {
                                System.out.println("String is Palindrome");
                                break;
                        }
                }
        }
}
```

© IETE

```
 public static void main(String []args)
 {
         Palimdrome palimObj = new Palimdrome();
         try
         {
                 String enteredString = palimObj.readData();
                 palimObj.checkPalindrome(enteredString);
         }
         catch(IOException ie)
         {
                 ie.printStackTrace();
         }
         catch(Exception e)
         {
                 e.printStackTrace();
         }
 }
}
```

Q6 (c) What are streams in Java? Explain.

Answer Page Number 281, 4th edition of textbook.

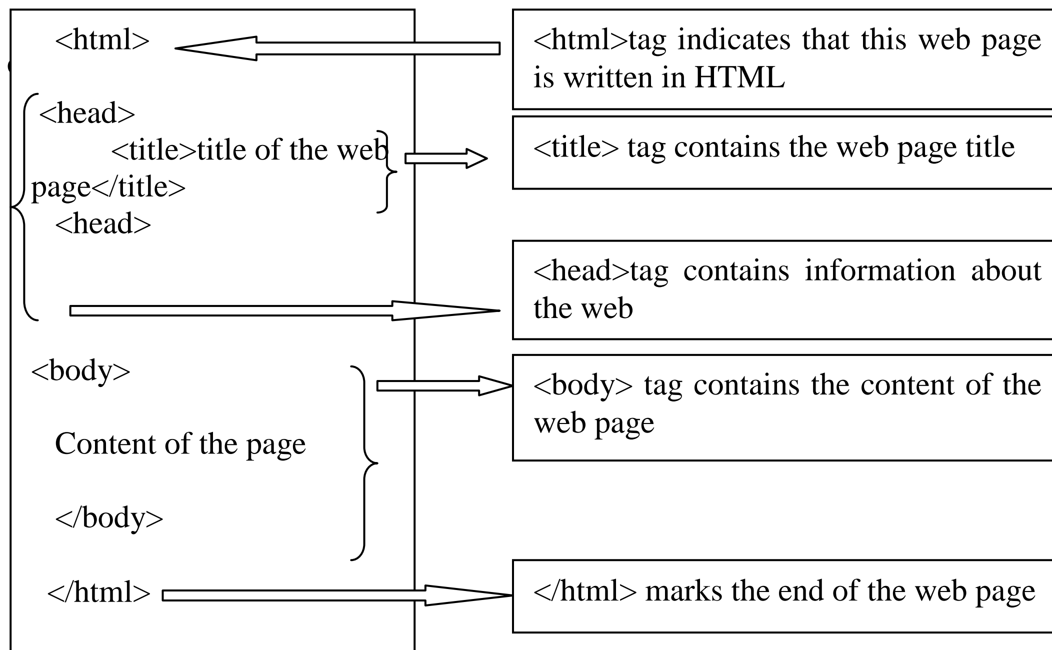
Q7 (a) What is HTML? Write the structure of an HTML document.

Answer
HTML is the "language" that web pages are written in - in fact, HTML stands for "hypertext mark-up language."
Hyper is the opposite of linear. Old-fashioned computer programs were necessarily linear - that is, they had a specific order. But with a "hyper" language such as HTML, the user can go anywhere on the web page at any time.
Text is just what you're looking at now - English characters used to make up ordinary words.

Mark-up is what is done to the text to change its appearance. For instance, "marking up" your text with before it and after it will put that text in bold.
Language is just that. HTML is the language that computers read in order to understand web pages.

Basic structure of an HTML document

An HTML document has two main parts:
Head: The head element contains title and meta data of a web document.
Body: The body element contains the information that you want to display on a web page.



DOCTYPE Declaration
The DOCTYPE declaration is the first part of coding that you should enter in your HTML document. This is required if you wish to validate your document with the W3C's validation service. Web browsers need to know what version of HTML/XHTML your page is written in to process the code correctly.
The HTML Tags
All HTML documents contain a <html> and </html> pair of tags. These tags identify the document's contents as HTML to the browser. The <html> tag goes in the line right under your DOCTYPE declaration. </html> is the last line of coding in your document.
Opening html tag:
<html>
The Head Tags - Opening Head Tag
The <head> and </head> tags identify the document's head area. The information between these two tags is not visible on your page.
Opening head tag:
<head>
Character Encoding
The character encoding meta tag tells the browser which character set the web page uses.
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
Title Tag
The title tag creates the page title that is seen in the title bar of the web page.
<title>Title of the document</title>
Meta Tags
The meta tags provide information about your web page.
<meta name="Description" content="Your description">
<meta name="Keywords" content="first, second, third">

```
<meta name="Author" content="Author Information">
<meta name="Copyright" content="Copyright Statement">
<meta name="Distribution" content="Global">
<meta name="Expires" content="Tue, 01 Jun 1999 19:58:02 GMT">
<meta name="Robots" content="index,follow">
```

Link Tag

The link tag is used to link other documents to this one.
This example shows linking to an external stylesheet.

```
<link rel="stylesheet" type="text/css" href="styles/stylesht.css">
```

Script Tag

The script tag defines what type of script the browser is to execute. This tag can also be included in the body of your page.

```
<script type="text/javascript">
 <!—
 <!--Your script -->
 -->
 </script>
```

Style Tag

The style tag is used to set the style of your document elements. It is better to use an external style sheet using the link tag so if you wish to change something you only have to change it in one spot.

```
<style type="text/css">
 Your style types
 </style>
```

Closing Head Tag

The closing head tag defines the end of the document's head section.

```
</head>
```

The Body Tags

The body tags surround the body (contents) of your web page.

```
<body>
 The body of the document
 </body>
```

Closing HTML Tag

The closing HTML tag is the last line in your HTML document. Don't put anything after this tag! Your page will not validate if you do.

```
</html>
```

Q7 (b)What is web hosting and types of web hosting services?

Web hosting is the service that makes your website available to be viewed by others on the Internet. A web host provides space on its server, so that other computers around the world can access your website by means of a network or modem.

Types of Web Hosting Services

Internet Service Providers (ISPs)

© IETE

Many people put up their first websites through their ISPs, because it's generally easy and inexpensive. Most ISP service packages include a small amount of free web space, along with tools to create and upload websites quickly and easily. ISP websites are perfect for people who want to put up small sites with low amounts of traffic. However, there are usually rate restrictions, and most ISPs don't offer a lot of features, so they might not be the best choice for a thriving business website.

Free Web Hosting

Free web hosting is another good option for smaller, personal websites. There are many free hosting providers that offer all types of features; some include CGI access and more. The drawback to most free hosting services is that they are funded by advertising that appears on your site, so free web hosting so generally best for personal, rather than business, websites.

Paid Hosting

With paid hosting, you pay a fee for space and services on a web hosting provider's server. Monthly fees can range from a few dollars to several hundred dollars. Obviously, the more you pay, the more features you should have at your disposal. Services can include CGI access, database support, ASP, e-commerce, SSL, additional space on the server, extra bandwidth, and more.

Domain Hosting

A good option for small businesses is to pay for domain hosting. Domain hosting allows you to host your site anywhere you like: on an ISP, a free hosting service, or even your own server. You buy a domain name and have the provider forward all requests for that domain to the actual web location. This is often less expensive than buying both the domain and the hosting service, and it allows businesses to brand their URLs.

Direct Internet Access

Hosting your site yourself offers you the most control over your web server. Companies with large data centers or that require high security in every aspect of their web and Internet access should look into this type of hosting.

Q7 (c) What is HTTP? Define purpose of any two HTTP methods.

Answer

HTTP, the Hypertext Transfer Protocol, is the application-level protocol that is used to transfer data on the Web. HTTP comprises the rules by which Web browsers and servers exchange information. Although most people think of HTTP only in the context of the World-Wide Web, it can be, and is, used for other purposes, such as distributed object management systems.

How Does HTTP Work?

HTTP Is a request-response protocol. For example, a Web browser initiates a request to a server, typically by opening a TCP/IP connection. The request itself comprises

- a request line,
- a set of request headers, and

- an entity.

The server sends a response that comprises

- a status line,
- a set of response headers, and
- an entity.

Purpose of the following http methods

The GET Method

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

A conditional GET method requests that the identified resource be transferred only if it has been modified since the date given by the If-Modified-Since header. The conditional GET method is intended to reduce network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring unnecessary data.

The GET method can also be used to submit forms. The form data is URL-encoded and appended to the request URI

The HEAD Method

A HEAD request is just like a GET request, except it asks the server to return the response headers only, and not the actual resource (i.e. no message body). This is useful to check characteristics of a resource without actually downloading it, thus saving bandwidth. Use HEAD when you don't actually need a file's contents.

The response to a HEAD request must never contain a message body, just the status line and headers.

A POST request is used to send data to the server to be processed in some way, like by a CGI script. A POST request is different from a GET request in the following ways:

There's a block of data sent with the request, in the message body. There are usually extra headers to describe this message body, like Content-Type: and Content-Length:

The request URI is not a resource to retrieve; it's usually a program to handle the data you're sending.

The HTTP response is normally program output, not a static file.

The most common use of POST, by far, is to submit HTML form data to CGI scripts. In this case, the Content-Type: header is usually application/x-www-form-url encoded, and the Content-Length: header gives the length of the URL-encoded form data. The CGI script receives the message body through STDIN, and decodes it.

Q8 (a) Explain briefly the border, cellpadding, cellspacing and background attributes of the table tag in XHTML.

Answer

Table Tag:

The <table> tag defines a table. Table headers, table rows, table cells and other tables can be put inside a <table> tag.

The “align” and “bgcolor” attributes of the <table> tag in HTML have been are not supported in XHTML.

Attributes of Table Tag

Border

Border establishes the size of the border surrounding the table. The default value is 0, which means there is no border at all. If border is without a value, the default is 1.

Example

<table border=2> </table>

Cellpadding

Cellpadding sets the amount of space between the cell walls and the contents. The default value for cellpadding is 1.

Example

<table border=2 cellpadding=4> </table>

Cellspacing

Cellspacing sets the amount of space between the cells of a table. If the borders are visible, cellspacing controls the width of internal borders.

Example

<table border=3 cellspacing=3> </table>

Width

Width sets the width of the table. It can be expressed either as an absolute value in pixels, or as a percentage of screen width.

Example

<table border=2 width=70%> </table>

Background

Background sets the background image for the table.
Example
<table border=1 background=”myimage.gif”> </table>

Bordercolor

Bordercolor sets the color of all the borders of the table.
Example
<table border=5 bordercolor=red> </table>

Q8 (b) Explain strict, transitional and frame set DTD (document type definition) in XHTML validation

Answer
XHTML Validation

An XHTML document is validated against a Document Type Definition.

An XHTML document is validated against a Document Type Definition (DTD). Before an XHTML file can be properly validated, a correct DTD must be added as the first line of the file.

The Strict DTD includes elements and attributes that have not been deprecated or do not appear in framesets:

!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"

The Transitional DTD includes everything in the strict DTD plus deprecated elements and attributes:

!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"

The Frameset DTD includes everything in the transitional DTD plus frames as well:

!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Frameset//EN"

Q8 (c) Give brief overview of the steps of information architecture (IA)

Answer

The six concrete steps to Information Architecture are:
(1) define goals
(2) define audience,
(3) create and organize content
(4) formulate visual presentation concepts
(5) develop site map and navigation
(6) design and produce visual forms.

Q9 (a) Give brief overview of CGI. What are the components of CGI model? Explain web server-handler interaction.

Answer

The **Common Gateway Interface (CGI)** is an essential tool for creating and managing comprehensive Web sites. With CGI, you can write scripts that create interactive, user-driven applications.

CGI is the part of the Web server that can communicate with other programs that are running on the server. With CGI, the Web server can invoke an external program, while passing user-specific data to the program (such as what host the user is connecting from, or input the user has supplied through an HTML form). The program then processes that data and the server passes the program's response back to the Web browser.

Components of the CGI Model

- Clients
    - Web browsers
- Web server
    - Mediates communication between browsers and handler programs
    - CGI protocol
        - Specifies interaction between
            - Browser and handler programs
              Function call syntax
            - Web server and handler programs
- Handler programs (or CGI scripts)
    - Any executables residing on the web server
    - Can be written in any language

© IETE

Web Server - Handler Interaction

- Information about a request comes from
    - Request line
    - Header line
    - Request body (POST)
- Handler input
    - Environment variables
    - Stdin(request body)
- Handler output

Stdout

Q9.b. Explain the use of the following array methods:

(i) pop()	**(ii) push(....)**
(iii) concat(....)	**(iv) reverse()**

Answer Page Number 321 of the textbook

<u>TEXT BOOKS</u>

1. Programming with Java- Primer, E. Balagurusamy, Third Edition, TMH, 2007.

2. An Introduction to Web Design + Programming, Paul S. Wang and Sanda S. Katila, Thomson Course Technology, India Edition, 2008.