Q.2 a. List and explain the three OOP principles. (4)

Answer:

The Three OOP Principles are:

Encapsulation: It is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

Inheritance: It is the process by which one object acquires the properties of another object.

Polymorphism: It is a feature that allows one interface to be used for a general class of actions.

Scheme: Listing: 1M

Explaining each concept: 1M each

b. Explain why the main function is declared as "public static void main(String args[])"? (4)

Answer:

Main function in Java is declared as "public static void main(String args[]).

- Public: Function should be accessible to outside the class.
- Static: This function can be called without creating the object of this class.
- Void: No return type.
- String Args[]: Command line arguments.

Scheme: Explaining each concept: 1M (1M x 4 = 4M)

c. List and explain any 8 Java Buzzwords. (8)

Answer:

Java Buzzwords:

Simple, Secure, Portable, Object-Oriented, Robust, Multithreaded, Architecture-Neutral, Interpreted, High Performance, Distributed & Dynamic.

Scheme: Any 8 buzzwords: 8M

Q.3 a. Explain the following keywords:

(i) Garbage Collection (ii) finalize() method (4)

Answer:

Garbage Collection: 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. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize() method: Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

Scheme: 2X 2 =4

b. Explain the different uses of Super keyword. (4)

Answer:

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.

Scheme: Explanation with syntax: 2 X 2 = 4M

c. Create an Interface called as Convert which has two functions convertDollors() and convertPounds(). Both function takes one argument and returns the amount into indian rupees. Write a class to implement the interface. (8)

Answer:

```
public interface Convert
{
public double convertDollors(double);
public double convertPounds(double);
}
public class RupeeExchange implements Convert
{
public double convertDollors(double)
{
…….
}
public double convertPounds(double)
{
…….
}
}
```

Scheme: Interface: 2M

Class: 4M

Main function & object creation: 2M

Q.4 a. Bring out the differences between throw & throws. (4)

Answer:

Throw: 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;

Throws: If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw.

type method-name(parameter-list) throws exception-list

{

// body of method

}

Scheme: Explaining each concept with example: 2 X 2 = 4M

b. Explain the different ways of creating threads in java with syntax. (4)

Answer:

Different ways of creating threads:

1. By extending thread class
2. By implementing Runnable interface

Scheme: Each Method with syntax: 2 X 2 = 4M

c. Write a Java class called as SavingsAccount with members as AccountNo and Balance. Provide depositAmount() & withdrawAmount() functions. If user tries to withdraw more money than his balance, then throw a User Defined Exception. (8)

Answer:

Scheme: SavingsBank class: 3M

Creating user defined exceptions: 3M

Main function & Object creation: 2M

Q.5 a. Compare byte stream class and character stream class. (4)

Answer:

Byte Streams and Character Streams

Java defines two types of streams: byte and character. Byte *streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte stream.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. This is why older code that doesn't use character streams should be updated to take advantage of them, where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. The byte stream classes are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II. Remember, to use the stream classes, you must import java.io.

The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data. Both methods are declared as abstract inside InputStream and OutputStream. They are overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes, Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes are shown in Table 13-2.

The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively. These methods are overridden by derived stream classes.

b. Write a program to illustrate BufferedInputStream and BufferedOutputStream. (4+4)

Answer:

BufferedInputStream

Buffering I/O is a very common performance optimization. Java's BufferedInputStream class allows you to "wrap" any InputStream into a buffered stream and achieve this performance improvement.

BufferedInputStream has two constructors:

BufferedInputStream(InputStream *inputStream*)
BufferedInputStream(InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk

block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the *host* operating system, the amount of memory available, and how the machine is configured. To make good use of buffering doesn't necessarily require quite this degree of sophistication. A good *guess* for a *size is* around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the **InputStream**, you will be manipulating fast memory most of the time.

Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the **read()** and **skip()** methods implemented in any **InputStream**, **BufferedInputStream** also supports the **mark()** and **reset()** methods. This support is reflected by **BufferedInputStream.markSupported()** returning **true**.

The following example contrives a situation where we can use **mark()** to remember where we are in an input stream and later use **reset()** to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset()** happens and where it does not.

```
// Use buffered input.
import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) throw IOException {
    String a = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = a.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marked = false;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;
        } else {
          marked = false;
        }
        braak;
      case ';':
        if (marked) {
          marked = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
```

```
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    }
}
```

Notice that this example uses **mark(32)**, which preserves the mark for the next 32 bytes read (which is enough for all entity references). Here is the output produced by this program:

```
This is a (c) copyright symbol but this is &copy not.
```

BufferedOutputStream

A **BufferedOutputStream** is similar to any **OutputStream** with the exception of an added **flush( )** method that is used to ensure that data buffers are physically written to the actual output device. Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.

Unlike buffered input, buffering output does not provide additional functionality. Buffers for output in Java are there to increase performance. Here are the two available constructors:

BufferedOutputStream(OutputStream *outputStream*)
BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

The first form creates a buffered stream using the default buffer size. In the second form, the size of the buffer is passed in *bufSize*.

PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. *Pushback* is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. The **PushbackInputStream** class implements this idea. It provides a mechanism to "peek" at what is coming from an input stream without disrupting it.

PushbackInputStream has the following constructors:

PushbackInputStream(InputStream *inputStream*)
PushbackInputStream(InputStream *inputStream*, int *numBytes*)

The first form creates a stream object that allows one byte to be returned to the input stream. The second form creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream.

Beyond the familiar methods of **InputStream**, **PushbackInputStream** provides **unread( )**, shown here:

void unread(int *ch*)
void unread(byte *buffer*[ ])
void unread(byte *buffer*, int *offset*, int *numChars*)

The first form pushes back the low-order byte of *ch*. This will be the next byte returned by a subsequent call to **read( )**. The second form returns the bytes in *buffer*. The third form pushes back *numChars* bytes beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a byte when the pushback buffer is full.

Here is an example that shows how a programming language parser might use a **PushbackInputStream** and **unread( )** to deal with the difference between the = = operator for comparison and the = operator for assignment:

```
// Demonstrate unread().
import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    PushbackInputStream f = new PushbackInputStream(in);
    int c;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '=':
        if ((c = f.read()) == '=')
          System.out.print(".eq.");
        else {
          System.out.print("<-");
          f.unread(c);
        }
        break;
      default:
        System.out.print((char) c);
        break;
      }
    }
  }
}
```

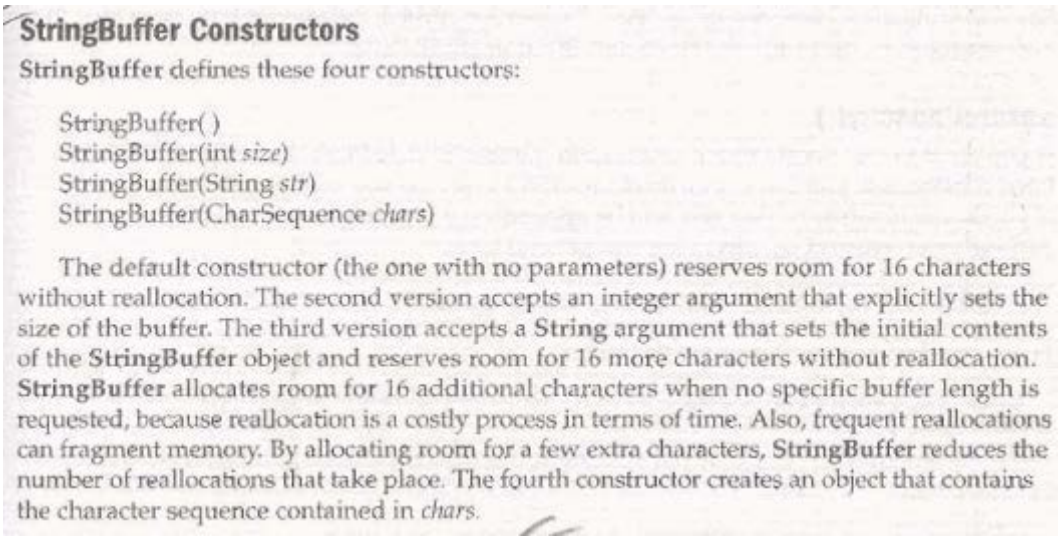
Here is the output for this example. Notice that = = was replaced by ".eq." and = was replaced by "<-".

```
if (a .eq. 4) a <- 0;
```

CAUTION *PushbackInputStream has the side effect of invalidating the* ***mark( )*** *or* ***reset( )*** *methods of the* ***InputStream*** *used to create it. Use* ***markSupported( )*** *to check any stream on which you are going to use* ***mark( )/reset( )****.*

c. Give the syntax of four StringBuffer constructors and give their application. (4)

Answer:



StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )
StringBuffer(int *size*)
StringBuffer(String *str*)
StringBuffer(CharSequence *chars*)

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars*.

Q.6 a. Bring out the differences between AWT & Swings. (4)

Answer:

SWINGS VS AWT:

AWT	SWINGS
AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
AWT components are **heavyweight**.	Swing components are **lightweight**.
AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
AWT provides **less components** than Swing.	Swing provides **more powerful components**such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Scheme: Any 4 difference: 4M

b. Explain any two layouts in AWT. (4)

Answer:

BorderLayout: The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.

CardLayout: The CardLayout object treats each component in the container as a card. Only one card is visible at a time.

FlowLayout: The FlowLayout is the default layout.It layouts the components in a directional flow.

GridLayout: The GridLayout manages the components in form of a rectangular grid.

GridBagLayout: This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.

Scheme: Any 2 Layouts with Syntax: 2 X 2 = 4M

c. Write a Java Swing program to create a UI which has a textbox and two buttons. One button will convert the text in the textbox from Upper case to Lower case and other resets the textbox. (8)

Answer:

Program:

```
import java.awt.event.*;
import javax.swing.*;

public class JTextFieldDemo extends JApplet {
        JTextField jtf1,jtf2;
        public void init() {
                try {
                        SwingUtilities.invokeAndWait(
                        new Runnable() {
                        public void run() {
                        makeGUI();
                }});
        }
        catch (Exception exc) {
                System.out.println("Can't create because of " + exc);
        }}
        private void makeGUI()
        {
                setLayout(new FlowLayout());
                jtf1 = new JTextField(15);
                add(jtf1);
                jtf2 = new JTextField(15);
                add(jtf2);
                JButton Convert = new JButton("Convert");
                add(Convert);
                Convert.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
        String data = jtf1.getText();
        String out  = data.toUpperCase();
        jtf2.setText(out);
        }
```

}

Scheme: 8M

PART B

Answer any TWO Questions. Each question carries 16 marks.

Q.7 a. What are the name servers? Explain (4)

Answer:

Name servers are the actual programs that provide the domain-to-IP mapping information on the Internet. We mentioned that DNS provides a distributed database service that supports dynamic retrieval of information contained in the name space. Web browsers, and other Internet client applications, will normally use the DNS to obtain the IP a target host before making contact with a server.

There are three elements to the DNS: the name space, the name servers, and the resolvers.

Scheme: Explanation: 4M

b. Bring out the differences between HTML & XHTML. (6)

Answer:

Following are the changes that XHTML brings compared to HTML.

- XHTML Documents Must be Well-Formed
- Elements and Attributes Must be in Lower Case
- End Tags are Required for all Elements
- Attribute Values Must Always be Quoted
- The Elements with id and name Attributes
- Attributes with Pre-defined Value Sets
- The <html> Element is a Must.

Scheme: Any 6 differences: 6M

c. List different concrete steps to Information Architecture. Explain how this can be helpful to reveal true objective of the site. (6)

Answer:

Steps for IA:

1. The six concrete steps to IA are :
2. Define goals
3. Define audience
4. Create and organize content
5. Formulates visual presentation concepts
6. Develop sitemap and navigation , and
7. Design and produce visual forms

Scheme: Explanation: 6M

Q.8 a. What is design? Explain the Elements of Design. (6)

Answer:

Design elements are the forms such as logos, icons, text blocks, and photos, included in a composition. The elements must be grouped and organized to create meaning in a given two-dimensional space. Most designers group these elements or forms according to a set of guidelines for two-dimensional design known as the design principles. So, elements are the objects, and principles are the guidelines for placing the objects in a layout or a particular arrangement of elements, known as the composition. Together, they can produce many different visual effects will discuss in this chapter such as contrast, hierarchy, focal points, unity, visual balance. These principles are said to be the \tools" of visual designers. But, as in any discipline, it is not enough to simply follow the rules. A designer needs to know how to create meaningful compositions. Understanding design principles is only the first step. The real learning begins when one immerses oneself in the design discipline and begins to see the world through different eyes.

Scheme: Explanation: 6M

b. Explain Webpage Layout Grids. (10)

Answer:

Scheme: Explanation of Layout Grids: 8M

Q.9 a. Describe the structure of CGI program. (8)

Answer:

A CGI program(or script) is invoked by the web server which provides input to the CGI program and receive its output . Typically, a CGI program follows this outline:

1. Determines request method and receives input data: The request method is given by the REQUST_METHOD environment variable. For a POST query the data is read from standard input and the length of the data is indicated by the CONTENT LENGTH environment variable .for a GET query, the input data is the value of the QUERY STRING environment variable.
2. Decodes and checks input data: the form –url encoded input data is decoded and the key- value pairs recoverse. The correctness of the input data is checked. Incomplete or incorrect input results in a response to the end-user for the correct information.
3. Performs tasks: The input data is complete and correct. The program now processes the information and perform required actions.
4. Produces output: A generated response is sent to standard output. The response is usually in HTML format.

Scheme: Explanation of each point: 2M (4 X 2 = 8M)

b. Explain different data types used in JavaScript. (8)

Answer:

JavaScript primitive data types

There are five types of primitive data types in JavaScript. They are as follows:

String: Represents sequence of characters

Number: Represents numeric values

Boolean: Represents boolean value either false or true
Undefined: Represents undefined value
Null: Represents null i.e. no value at all

JavaScript non-primitive data types
The non-primitive data types are as follows:
Object: Represents instance through which we can access members
Array: Represents group of similar values
RegExp: Represents regular expression

Scheme: Primitive Types: 4M
Non-primitive types: 4M

TEXT BOOK

I. The Complete Reference Java, Herbert Schildt, TMH, Seventh Edition, 2007
II. An Introduction to Web Design + Programming, Paul S. Wang and Sanda S. Katila, Thomson Course Technology, India Edition, 2008