Q.1 a. The signature for *main()* function in every Java application program is *public static void main(String args[])*
What is the role of keyword *'static' in the* above declaration? What happens if we do not write keyword *static* in the above declaration?

Answer:
The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made.
If we omit the keyword static from the main function declaration, the program will compile but there will be runtime error such as *"Exception in thread "main" java.lang.NoSuchMethodError : Main"*.

b. What is method overloading? Is return type alone sufficient for method overloading?

Answer:
In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java implements polymorphism. Method overloading is one of Java's most exciting and useful features.
When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.
No, only return type of the method is not sufficient for method overloading.
While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

c. Under what circumstances does a compiler insists that a class must be declared as an abstract class? Can an abstract class be declared as final also?

Answer:
The compiler insists that a class must be declared as an abstract class if any of the following conditions is true:

(i) the class has one or more abstract methods
(ii) the class inherits one or more abstract methods (from an abstract parent) for which it does not provide implementation.
(iii) The class declares that it implements an interface but does not provide implementations for every method of that interface.

Thus, according to these three conditions, the abstract class is in some sense is incomplete. Thus, this class must be subclassed. But a class declared as final class can't be subclassed. Hence, a class cannot be declared as abstract and final simultaneously.

d. List the differences between Java application and Java applet?

Answer:
There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

e. What is Anonymous Inner Classes? Illustrate how an Anonymous Inner Class can facilitate the writing of event handlers?

Answer:
An *anonymous* inner class is one that is not assigned a name. Consider the applet shown in the following listing. As before, its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet {
        public void init() {
                addMouseListener(new MouseAdapter() {
                        public void mousePressed(MouseEvent me) {
                                showStatus("Mouse Pressed");
                        }
                });
        }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully.

The syntax **new MouseAdapter( ) { ... }** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly.

f. Java has a standard class called MouseEvent. What is the purpose of this class? What does an object of type MouseEvent do?

Answer:

When an event occurs, the system packages information about the event into an object. That object is passed as a parameter to the event-handling routine. Different types of events are represented by different classes of objects. An object of type MouseEvent represents a mouse or mouse motion event. It contains information about the location of the mouse cursor and any modifier keys that the user is holding down. This information can be obtained by calling the instance methods of the object. For example, if evt is a MouseEvent object, then evt.getX() is the x-coordinate of the mouse cursor, and evt.isShiftDown() is a boolean value that tells you whether the user was holding down the Shift key.

g. Java uses "Garbage collection" for memory management. Explain what is meant here by Garbage collection. What is the alternative to Garbage collection?

Answer:

The purpose of garbage collection is to identify objects that can no longer be used, and to dispose of such objects and reclaim the memory space that they occupy. If garbage collection is not used, then the programmer must be responsible for keeping track of which objects are still in use and disposing of objects when they are no longer needed. If the programmer makes a mistake, then there is a "memory leak," which might gradually fill up memory with useless objects until the program crashes for lack of memory.

Q.2 a. With the help of a Java program explain the difference between "equals and ==".

Answer:

The **equals( )** method and the == operator perform two different operations. 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:

```
// equals() vs ==
    class EqualsNotEqualTo {
        public static void main(String args[]) {
            String s1 = "Hello";
            String s2 = new String(s1);
            System.out.println(s1 + " equals " + s2 + " -> " +
                s1.equals(s2));
            System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
        }
    }
```

The variable **s1** refers to the **String** instance created by "**Hello**". The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not **==**, as is shown here by the output of the preceding example:

OUTPUT

```
Hello equals Hello -> true
Hello == Hello -> false
```

b. Explain the use of final keyword in respect of inheritance.

Answer:

The uses of **final** keyword applicable to inheritance are as follows:

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. 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!");
    }
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

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 runtime. 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**.

c. Suppose that a class, *Employee,* is defined as follows:

```
class Employee {
String lastName;
String firstName;
double hourlyWage;
int yearsWithCompany;
}
```

Suppose that data about 100 employees is already stored in an array:

```
Employee[] employeeData = new Employee[100];
```

Write a code segment that will output the first name, last name and hourly wage of each employee who has been with the company for 20 years or more..

Answer:

(The data for the i-th employee is stored in an object that can be referred to as employeeData[i]. The four pieces of data about that employee are members of this object and can be referred to as:

```
employeeData[i].firstName
employeeData[i].lastName
employeeData[i].hourlyWage
employeeData[i].yearsWithCompany
```

The code segment uses a for loop to consider each employee in the array.)

```
for (int i=0; i < 100; i++) {
```

```
            if ( employeeData[i].yearsWithCompany >= 20 )
                System.out.println(employeeData[i].firstName + " " +
                    employeeData[i].lastName + ": " +
                    employeeData[i].hourlyWage);
    }
```

Q.3 a. Define Interface. How is it similar to and different from a class? Can an interface be extended? If yes, explain with the help of an example?

Answer:

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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;
}
```

Here, *access* is either **public** or not used. 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. *name* is the name of the interface, and can be any valid identifier. The methods which are declared within the interface have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Here is an example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback {
        void callback(int param);
}
```

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

An interface is different from a class in several ways, such as:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
        interface A {
                void meth1();
                void meth2();
        }

// B now includes meth1() and meth2() -- it adds meth3().
        interface B extends A {
                void meth3();
        }
// This class must implement all of A and B

        class MyClass implements B {
                public void meth1() {
                        System.out.println("Implement meth1().");
                }
                public void meth2() {
                        System.out.println("Implement meth2().");
                }
                public void meth3() {
                        System.out.println("Implement meth3().");
                }
        }
```

```
class IFExtend {
        public static void main(String arg[]) {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

If you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

b. Java has two ways to create Child threads. What are these two methods? Explain them. Which of the two do you think is better and why?

Answer:

Java defines two methods to create a child thread. These are

(i) By implementing the **Runnable** interface.

(ii) By extending the **Thread** class, itself.

The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement **Runnable** interface, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, we define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After the class has been created that implements **Runnable**, we will instantiate an object of type **Thread** from within that class. After the new thread is created, it will not start running until the method **start( )** has been called , which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**.

The **start( )** method is shown here

void start( )

Here is the program that will show how to create a child thread by implementing the interface **Runnable**.

```
class NewThread implements Runnable {
        Thread t;
        NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
        }
```

```
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}

class ThreadDemo {
        public static void main(String args[]) {
                new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.
Here is the program that will show how to create a child thread by extending class **Thread**.

```
class NewThread extends Thread {
        NewThread() {
                // Create a new, second thread
                super("Demo Thread");
                System.out.println("Child thread: " + this);
                start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
```

```
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}

class ExtendThread {
        public static void main(String args[]) {
                new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run( )**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.

Q.4 a. What are Packages? How are Packages declared and defined in Java programming language? Write the advantages of using Packages in Java programs.

Answer:

Java provides a mechanism for partitioning the class name space into more manageable mechanism by means of the package. The package is both a naming and a visibility control mechanism. Classes can be defined inside a package that is not accessible by code outside that package. You can also define class members that are only exposed to other

members of the same package. This allows the classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.
Thus, a **package** can be defined as " a collection of related classes and interfaces that provides access protection and namespace management".

Creating a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. While the default package is fine for small and simple programs, it is inadequate for real applications. Most of the time, you will define a package for your code.
The general form of the **package** statement:
package *pkg*;
Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**.
package MyPackage;
Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. It is case sensitive thus the directory name must match the package name exactly.
More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.
You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
package *pkg1*[.*pkg2*[.*pkg3*]];
A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as
package java.awt.image;
needs to be stored in **java/awt/image**, **java\awt\image**, or **java:awt:image** on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Advantages of using Packages in Java program:

- It makes classes easier to find and use.
- It helps in avoiding naming conflicts.
- It helps in controlling the access.

b. Create a try block that is likely to generate exception at three different places. Provide the necessary catch blocks to catch and handle those exceptions.

Answer:

```
class ThreeException {
        static void threeExcep(int a) {
                try {
                        /* If one command-line arg is used, then a divide-by-zero
                exception will be generated by the following code. */
                        if(a==1) a = a/(a-a); // division by zero
                        /* If two command-line args are used, then generate an out-of-
                bounds exception. */
                        if(a==2) {
                                int c[] = { 1 };
                                c[42] = 99; // generate an out-of-bounds exception
                        }
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index out-of-bounds: " + e);
        }
}
        public static void main(String args[]) {
                try {
                        int a = args.length;
                                /* If no command-line args are present, the following
                        statement will generate a divide-by-zero exception. */
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        threeExcep(a);
                } catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
        }
}
```

Q.5 a. What is Event Delegation Model? Explain the terms event, event source and event listener in this respect?

Answer:

The modern approach of handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this event model, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle.

Event

An *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

Event Source

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. The general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Here, *Type* is the name of the event and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive

notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

b. Write a program in Java that reads a string and rewrite it in the alphabetical order. For example, the word STRING should be written as GINRST.

Answer:

```
 import java.io.*;
public class StringSorting{

        public static void main(String args[]){
                String name = null;
                try {
                        System.out.print("Enter string\n");
                        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                        name = br.readLine();
                        char nameChar[] = new char[100];

                        nameChar = name.toCharArray();


                        nameChar = name.toCharArray();

                for(int i=0;i<nameChar.length;i++){
                        for(int j=0;j<nameChar.length-i-1;j++){
                                        if(nameChar[j] >nameChar[j+1]){
                                        char temp = nameChar[j];;
                                        nameChar[j] = nameChar[j+1];
                                        nameChar[j+1] = temp;
                                }
                        }
                }

                System.out.print("\n Entered String's character in sorted order are\n");
                for(int k = 0;k<nameChar.length;k++){
                        System.out.print(nameChar[k]);
                }

                }
                catch(IOException ex){
```

```
                    System.out.println("NavError = " + ex);
                }
            }
}
```

Q.6 a. Suppose that you are writing an applet and you want the applet to respond in some way when the user clicks the mouse on the applet. What are the four things you need to remember to put into the source code of your applet?

Answer:

(1) Since the event and listener classes are defined in the java.awt.event package, you have to put "import java.awt.event.*;" at the beginning of the source code, before the class definition.

(2) The applet class must be declared to implement the MouseListener interface, by adding the words "implements MouseListener" to the heading. For example: "public class MyApplet extends JApplet implements MouseListener". (It is also possible for another object besides the applet to listen for the mouse events.)

(3) The class must include definitions for each of the five methods specified in the MouseListener interface. Even if a method is not going to do anything, it has to be defined, with an empty body.

(4) The applet (or other listening object) must be registered to listen for mouse events by calling addMouseListener(). This is usually done in the init() method of the applet.

b. Explain what is meant by *input focus?* How is the input focus managed in a Java GUI program?

Answer:

When the user uses the keyboard, the events that are generated are directed to some component. At a given time, there is just one component that can get keyboard events. That component is said to have the input focus. Usually, the appearance of a component changes if it has the input focus and wants to receive user input from the keyboard. For example, there might be a blinking text cursor in the component, or the component might be hilited with a colored border. In order to change its appearance in this way, the component needs to be notified when it gains or loses the focus. In Java, a component gets this notification by listening for focus events.

Some components, including applets and canvasses, do not get the input focus unless they request it by calling requestFocus(). If one of these components needs to process keyboard events, it should also listen for mouse events and call requestFocus() when the user clicks on the component. (Unfortunately, this rule is not enforced uniformly on all platforms.)

c. What are the different methods that are used to play an audio file represented by the AudioClip interface in the java.applet package? Use those methods in your Java applet to illustrate the steps to play an audio.

Answer:

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including:

- **public void play():** Plays the audio clip one time, from the beginning.
- **public void loop():** Causes the audio clip to replay continually.
- **public void stop():** Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is the example showing all the steps to play an audio:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet {
  private AudioClip clip;
  private AppletContext context;
  public void init()    {
    context = this.getAppletContext();
    String audioURL = this.getParameter("audio");
    if(audioURL == null)      {
      audioURL = "default.au";
    }
    try {
      URL url = new URL(this.getDocumentBase(), audioURL);
      clip = context.getAudioClip(url);
    } catch(MalformedURLException e) {
      e.printStackTrace();
      context.showStatus("Could not load audio file!");
    }
  }
  public void start() {
    if(clip != null) {
      clip.loop();
    }
  }
  public void stop() {
    if(clip != null) {
      clip.stop();
    }
  }
}
```

Now let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="AudioDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
</html>
```

Q.7 a. What are Java Beans? Give its advantages? What are the various steps needed to create a new Bean?

Answer:

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components. Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Java Beans

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Here are the steps that must be followed to create a new Bean:

1. Create a directory for the new Bean.
2. Create the Java source file(s).
3. Compile the source file(s).
4. Create a manifest file.
5. Generate a JAR file.
6. Start the BDK.
7. Test.

b. What are Datagrams? How does Java implement Datagrams? Write the use of the following methods for accessing the internal state of a Datagram packet:
(i) InetAddress getAddress()
(ii) int getPort()
(iii) byte[] getData()
(iv) int getLength()

Answer:

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: The **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s.

DatagramPacket defines several constructors. The first constructor specifies a buffer that will receive data, and the size of a packet. It is used for receiving data over a **DatagramSocket**. The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Here are the four constructors:

DatagramPacket(byte *data*[ ], int *size*)
DatagramPacket(byte *data*[ ], int *offset*, int *size*)
DatagramPacket(byte *data*[ ], int *size*, InetAddress *ipAddress*, int *port*)
DatagramPacket(byte *data*[ ], int *offset*, int *size*, InetAddress *ipAddress*, int *port*)

There are several methods for accessing the internal state of a **DatagramPacket**. They give complete access to the destination address and port number of a packet, as well as the raw data and its length.

(i) InetAddress getAddress( ) : Returns the destination **InetAddress**, typically used for sending.
(ii) int getPort( ) : Returns the port number.
(iii) byte[ ] getData( ) : Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
(iv) int getLength( ) : Returns the length of the valid data contained inthe byte array that would be returned from the **getData( )** method. This typically does not equal the length of the whole byte array.

Text Books

1. Cay Horstmann - Computing Concepts with Java 2 Essentials, John Wiley, 3rd edition.
2. E. Balagurusamy- Programming with Java: A Primer, 3rd Edition, 2006, Tata McGraw-Hill.