CHAPTER

3

PACKAGES

CHAPTER OUTLINE

3.1 PACKAGE

3.2 ABOUT JAVA API PACKAGES

3.3 CONCEPT OF CLASS PATH

3.4 CONCEPT OF ACCESS SPECIFIERS

3.5 CONCEPT OF CREATING ACCESSING AND USING A PACKAGE AND SUBPACKAGE

3.6 APPRECIATE THE CONCEPT OF IMPORTING PACKAGES

3.7 EXPLORING IO AND UTIL PACKAGES

3.8 VARIOUS STREAM CLASSES

3.1 PACKAGE

DEFINITION

Package is a group of similar types of classes, interfaces and sub-packages. Package is a way of grouping variety of classes together. In fact packages act as "Containers" for classes.

Package can be categorized into two types. They are :

1. **Built-in Packages :** Predefined packages in java are those which are developed by developers. They are also called ***built-in packages*** in java.

   These packages consist of a large number of predefined classes, interfaces, and methods that are used by the programmer to perform any task.

2. **User-defined Packages :** The package which is defined by the user is called a ***User-defined package.*** It contains user-defined classes and interfaces.

Advantages of Package :

(i) Package is used to categorize the classes and interfaces.

(ii) Package provides access protection.

(iii) Package removes naming collision.

(iv) Package helps to improve code reusability.

3.2 ABOUT JAVA API PACKAGES

PREDEFINED PACKAGES/BUILT-IN PACKAGES/JAVA API PACKAGES

Predefined packages in java are those which are developed by Sun Microsystem. They are also called ***built-in packages in java.***

These packages consist of a large number of predefined classes, interfaces, and methods that are used by the programmer to perform any task.

Java APIs contains the following predefined packages, as shown in Fig. 3.1.

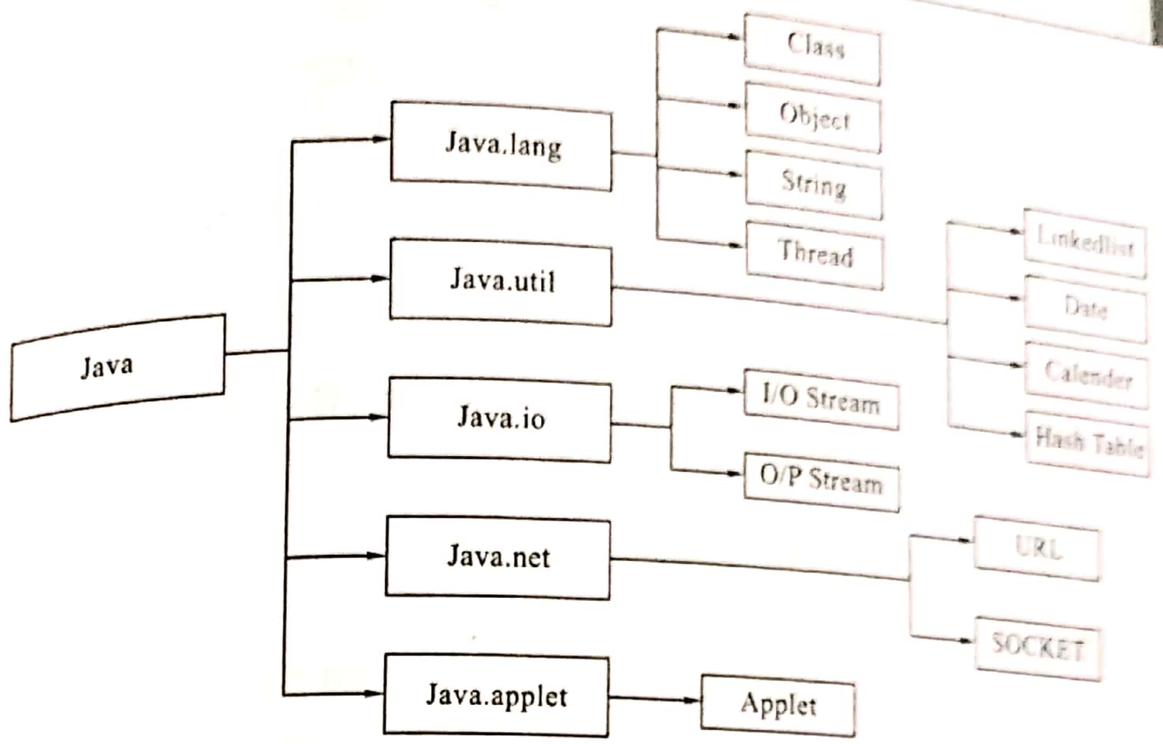


FIG 3.1 : Predefined Packages in Java

CORE PACKAGES

1. **Java.lang :** lang stands for language. The Java language package consists of java classes and interfaces that form the core of the Java language and the JVM. It is a fundamental package that is useful for writing and executing all Java programs.

   Eg : classes, objects, String, Thread, predefined data types, etc., It is imported automatically into the Java programs.

2. **Java.io :** io stands for input and output. It provides a set of I/O streams that are used to read and write data to files. A stream represents a flow of data from one place to another place.

3. **Java util :** Util stands for utility. It contains a collection of useful utility classes and related interfaces that implement data structures like LinkedList, Dictionary, HashTable, stack, vector, Calender, data utility, etc.,

4. **Java.net :** net stands for network. It contains networking classes and interfaces for networking operations. The programming related to client-server can be done by using this package.

WINDOW TOOLKIT AND APPLET

1. **Java.awt :** awt stands for abstract window toolkit. The Abstract window toolkit packages contain the GUI(Graphical User Interface) elements such as buttons, lists, menus, and text areas. Programmers can develop programs with colorful screens, paintings, and images, etc., using this package.

2. **Java.awt.image :** It contains classes and interfaces for creating images and colors.
3. **Java.applet :** It is used for creating applets. Applets are programs that are executed from the server into the client machine on a network.
4. **Java.text :** This package contains two important classes such as DateFormat and NumberFormat. The class DateFormat is used to format dates and times. The NumberFormat is used to format numeric values.
5. **Java.sql :** SQL stands for the structured query language. This package is used in a Java program to connect databases like Oracle (or) Sybase and retrieve the data from them.

3.3 CONCEPT OF CLASS PATH

CLASS PATH

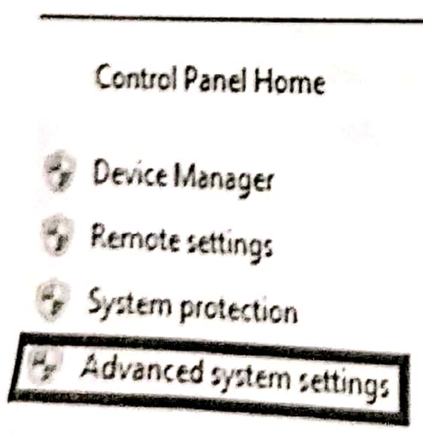
Class path is an environment variable which is used by Application class loader to locate and load the .class files. The class path defines the path, to find third-party and user-defined classes that are not extensions (or) part of Java platform. Include all the directories which contain .class files and JAR files when setting the class path.

There are two methods to set class paththrough command prompt (or) by setting environment variable.

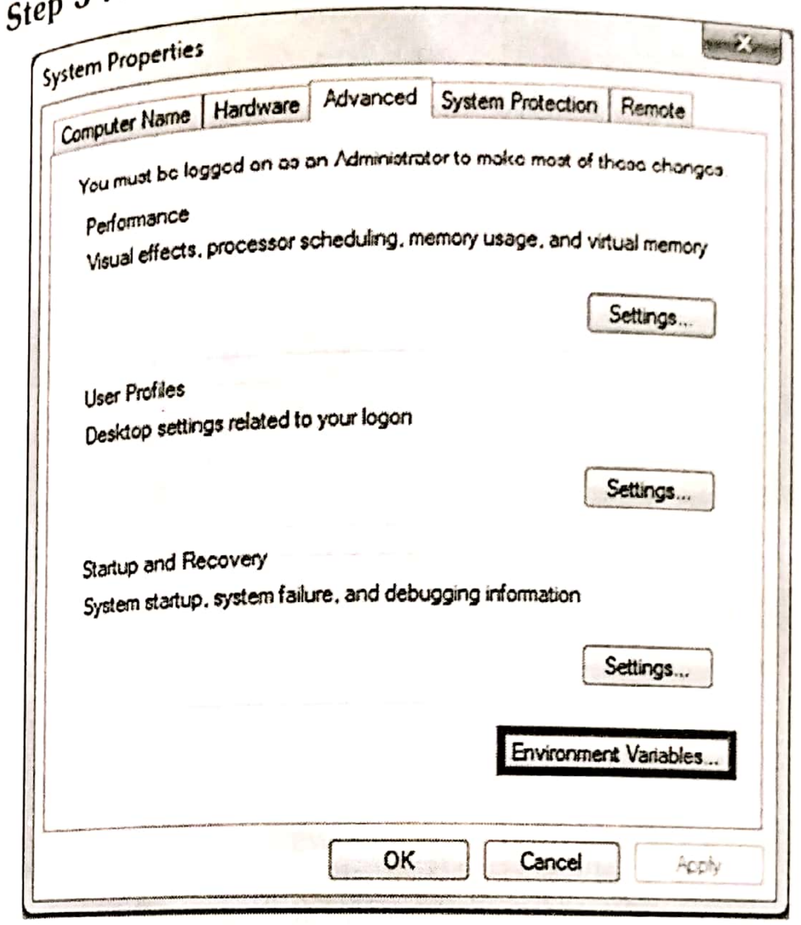
Let's see first method to set class path of MySQL database by environment variable.

Step 1 : Click on the Windows button and choose Control Panel. Select System.

Step 2 : Click on Advanced System Settings.



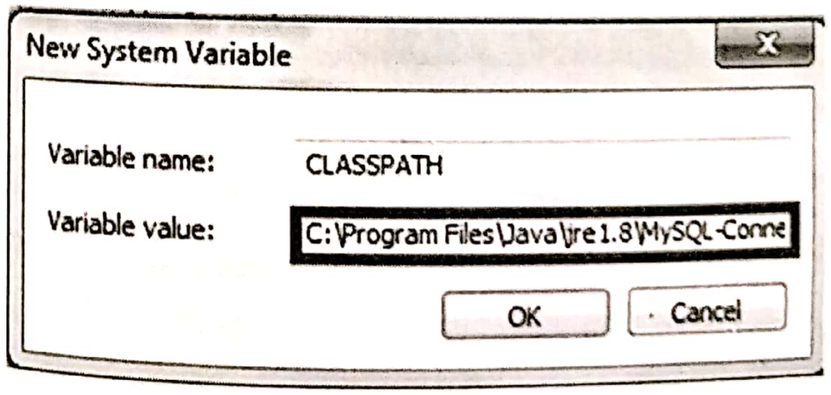
Step 3 : A dialog box will open. Click on Environment Variables.



Step 4 : If the class path already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the class path doesn't exist in System Variables, then click on the New button and type Variable name as class path and Variable value as C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.;

Remember : Put ;.; at the end of the class path.



Let us see second method to Setclass path in windows using command prompt

Type the following command in your Command Prompt and press enter.

```
setCLASS PATH=%CLASS PATH%;C:\Program Files\Java\jre1.8\rt.jar;
```

In the above command, The set is an internal DOS command that allows the user to change the variable value. CLASS PATH is a variable name. The variable enclosed in percentage sign (%) is an existing environment variable. The semicolon is a separator, and after the (;) there is the PATH of rt.jar file.

HOW TO SET PATH

The path is required to be set for using tools such as javac, java, etc.,

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you want to store your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are Two Ways to Set the Path in Java :

1. Temporary.
2. Permanent.

1. First method to set the Temporary Path of JDK in Windows.

   To set the temporary path of JDK, you need to follow the following steps :

   - Open the command prompt.
   - Copy the path of the JDK/bin directory.
   - Write in command prompt: set pat.
   - h=copied_path.

   For Example :

   ```
   set path=C:\Program Files\Java\jdk1.6.0_23\bin
   ```

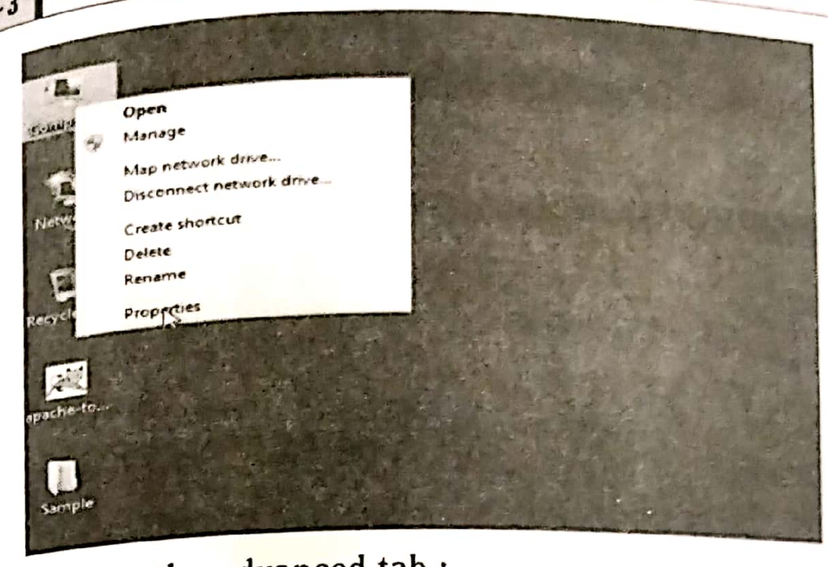
2. Second method to set Permanent Path of JDK in Windows.

   For setting the permanent path of JDK, you need to follow these steps :

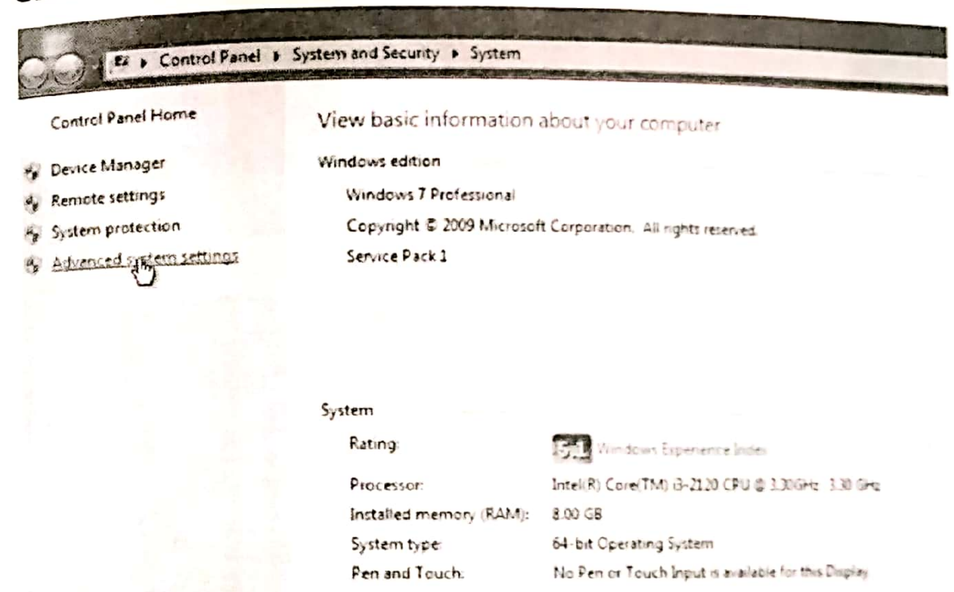
   · Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

   For Example :

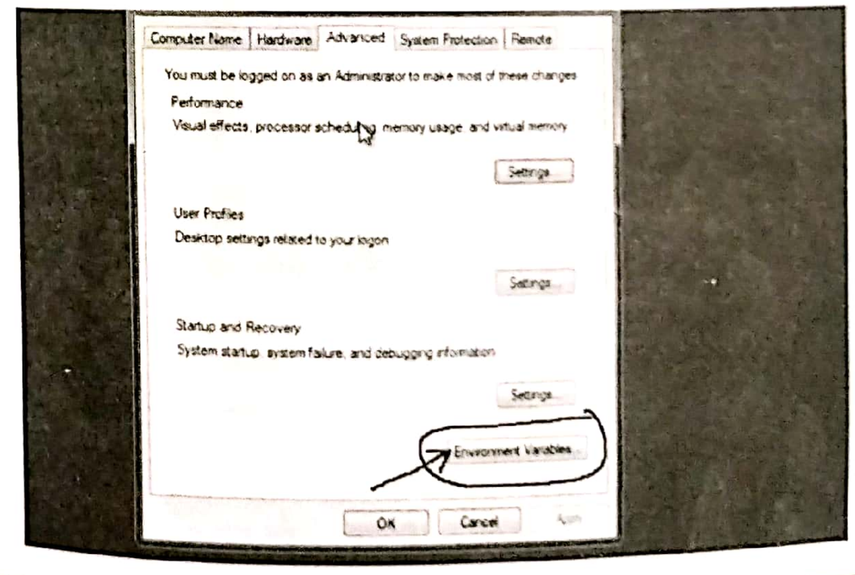
   (i) Go to MyComputer Properties.



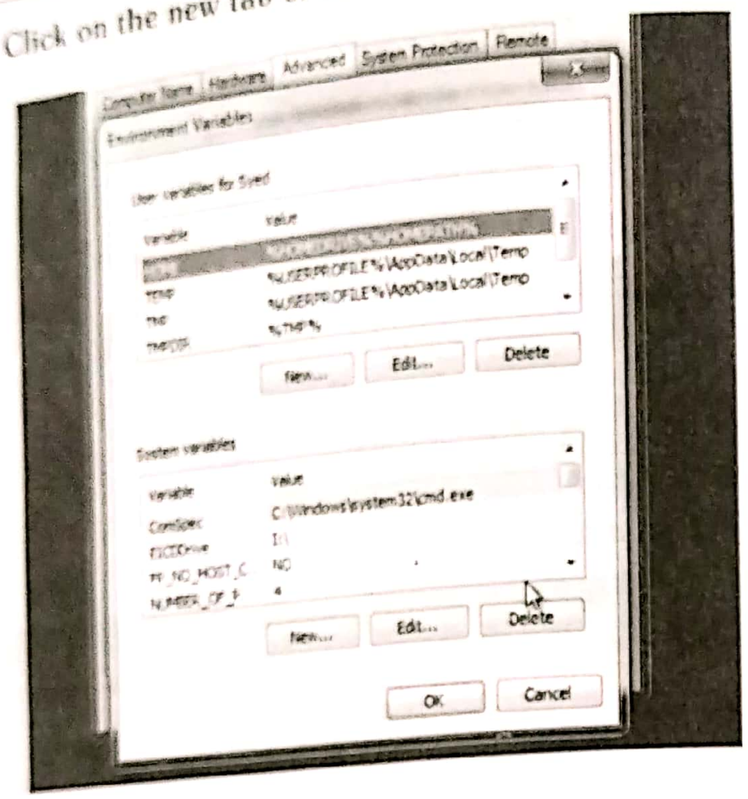
(ii) Click on the advanced tab :



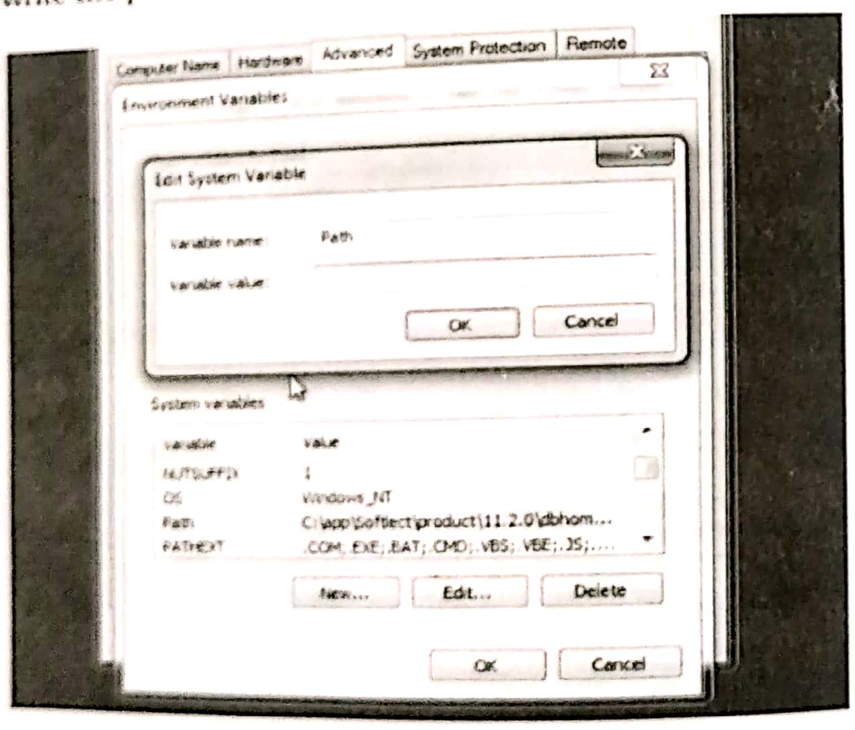
(iii) Click on environment variabless :



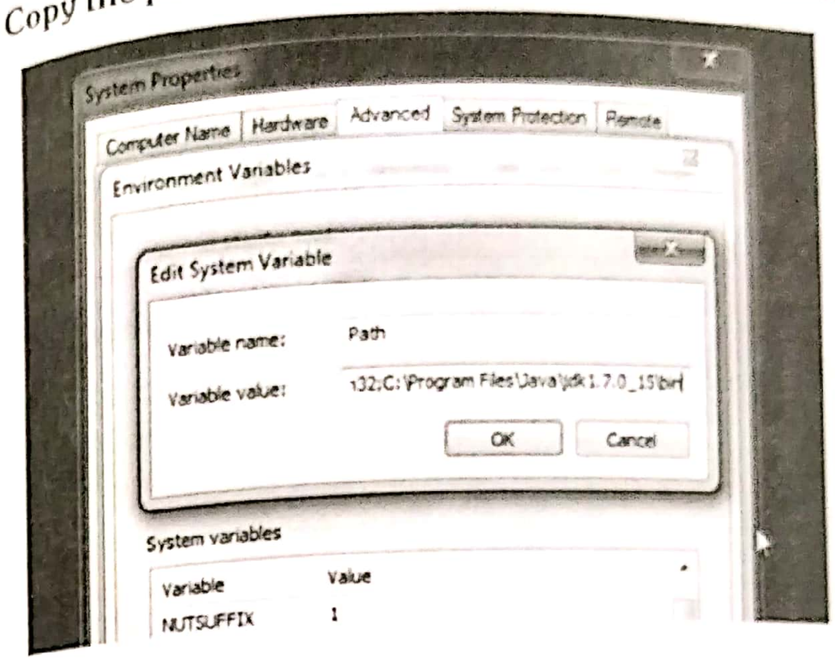
(iv) Click on the new tab of user variables :



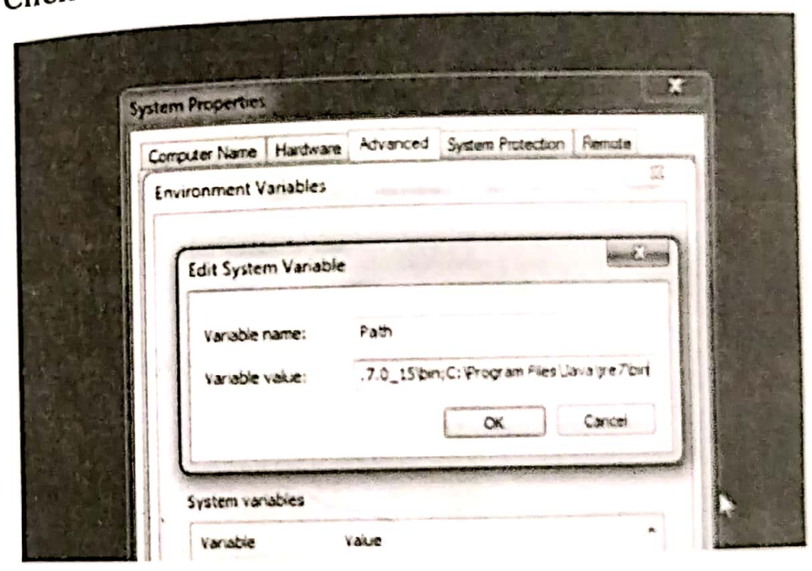
(v) Write the path in the variable name :



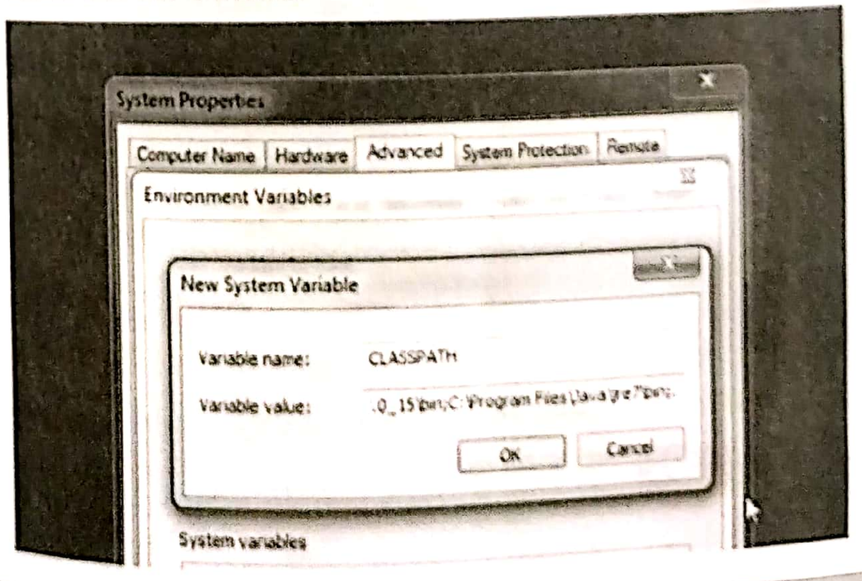
(vi) Copy the path of bin folder and Paste path of bin folder in the variable value.



(vii) Click on ok button.



(viii) Click on ok button.



Now your permanent path is set. You can now execute any program of java from any drive.

Differences between PATH and CLASS PATH :

S.No.	Path	Class Path
1.	PATH is an environment variable.	CLASS PATH is also an environment variable.
2.	It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
3.	You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
4.	PATH environment variable once set, cannot be overridden.	The CLASS PATH environment variable can be overridden by using the command line option -cp (or) -CLASS PATH to both javac and java command.

3.4 CONCEPT OF ACCESS SPECIFIERS

ACCESS MODIFIERS/ACCESS SPECIFIERS

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor (or) class.

There are 4 Types of Java Access Modifiers :

1. Private
2. Default
3. Protected
4. Public.

1. **Private** : The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default** : The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected** : The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public** : The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Let's understand the access modifiers in Java by a simple table.

EXAMPLE-2

Java program using private access modifier.

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members inside the class.

```
class A
{
private int data=40;
private void msg()
{
System.out.println("Hello java");
}
 public static void main(String args[])
  {
   A obj=new A();
   System.out.println(obj.data);
   obj.msg();
   }
}
```

Output :

```
40
Hello java
```

2. **Private Constructor :** If you make any class constructor private, you cannot create the instance of that class from outside the class.

EXAMPLE :

```
class A
{
private A()   //private constructor
{ }
void msg()
{
System.out.println("Hello java");
```

```
class B
{
 public static void main(String args[])
 {
  A obj = new A();//Compile Time Error
  obj.msg();//Compile Time Error
 }
}
```

Output :

```
compile-time error
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

EXAMPLE-2

Java program using default access modifier.

In this example, we have created one package i.e pack1. We are accessing the msg() of A class with in the package.

```
//save by A.java
package pack1;
class A
{
  void msg()
  {
System.out.println("Hello");
  }
}
  public static void main(String args[])
  {
   A obj = new A();
   obj.msg();
  }
}
```

Output : Hello

4. **Protected :** The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

EXAMPLE-1

Java program using protected access modifier.

In this example, we have created the two packages pack2 and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg() method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack2;
public class A
{
protected void msg()
  {
System.out.println("Hello");
  }
}
//save by B.java
package mypack;
import pack2.*;
class B extends A
{
  public static void main(String args[])
  {
   B obj = new B();
   obj.msg();
  }
}
```

Output : Hello

5. **Public :** The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

EXAMPLE :

Public access modifier.

```
//save by A.java
package pack;
public class A
{
public void msg()
 {
System.out.println("Hello");
 }
}
//save by B.java
package mypack;
import pack.*;
class B
{
 public static void main(String args[])
 {
  A obj = new A();
  obj.msg();
 }
}
```

Output : Hello

3.5 CONCEPT OF CREATING ACCESSING AND USING A PACKAGE AND SUBPACKAGE

To create a package we use package keyword

Syntax to create a package is given below.

```
package packagename ;
```

Example :

```
package p1;
```


Java uses file system directories to store packages. .class file with in a package is stored in directory which is same name as package.

Package statement is the first statement in any Java file.

EXAMPLE-1

Java program on creation of package concept.

```
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
    A obj=new A();
obj.msg();
 }
}
```

There are three ways to access the package from outside the package :

1. importpackagename.*;
2. import packagename.classname;
3. fully qualified name.

I. **Using packagename.* :** If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

EXAMPLE

Package that import the packagename.*

```
//save by A.java
package p1;
```

```
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();s
obj.msg();
 }
}
```

```
//save by B.java
package p2;
import p1.*;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
    B obj2=new B();
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```


2. **Using packagename.classname :** If you import package.classname then only declared class of this package will be accessible.

EXAMPLE

Package by import package.classname.

```
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
```

```
//save by B.java
package p2;
import p1.A;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
    B obj2=new B();
```

```
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```

3. **Using Fully Qualified Name :** If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to use import. But you need to use fully qualified name every time when you are accessing the class (or) interface.

EXAMPLE

Package by import fully qualified name.

```
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
//save by B.java
package p2;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
```



```
 }
public static void main(String args[])
 {
p1.Aobj=new p1.A();
obj.msg();
   B obj2=new B();
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```

Note : If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

4. **Subpackage :** Package inside the package is called the *subpackage*. It should be created to categorize the package further.

EXAMPLE

Subpackage.

```
//The below code illustrate creation of package
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
```

```
obj.msg();
  }
}
//The below code illustrate creation of sub-package
//save by A1.java
package p1.subpack;
public class A1
{
public void msg1()
 {
System.out.println("I HAVE PASSED TENTH CLASS");
 }
public static void main(String args[])
 {
   A1 obj1=new A1();
obj1.msg1();
 }
}
//save by B.java
package p2;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
p1.Aobj=new p1.A();
obj.msg();
A1 obj1=new A1();
obj1.msg1();
   B obj2=new B();
```



```
obj2.display();
  }
}
```

To Compile: javac -d . B.java

To Run: java p2.B

Output :

```
MY NAME IS ANIL
I HAVE PASSED TENTH CLASS
I AM STUDYING DCME FINAL YEAR
```

3.6 APPRECIATE THE CONCEPT OF IMPORTING PACKAGES

To access one package properties into another package, We have to import the package.

To import any package we use import keyword.

Syntax to import the package is given below.

```
import packagename1[.packagename2][.packagename3]...(classname/*);
```

For example :

```
import  p1.*;
```

There are three ways to access the package from outside the package :

1. importpackagename.*;
2. import packagename.classname;
3. fully qualified name.

1. **Using packagename.* :** If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

EXAMPLE :

Package that import the packagename.*

```
//save by A.java
package p1;
public class A
{
```

```
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
//save by B.java
package p2;
import p1.*;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
    B obj2=new B();
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```

2. **Using packagename.classname :** If you import package.classname then only declare class of this package will be accessible.

EXAMPLE

package by import package.classname.

```
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
//save by B.java
package p2;
import p1.A;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
    B obj2=new B();
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```

3. **Using fully qualified name :** If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to use import. But you need to use fully qualified name every time when you are accessing the class (or) interface.

EXAMPLE

package by import fully qualified name.

```
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
//save by B.java
package p2;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
p1.Aobj=new p1.A();
obj.msg();
   B obj2=new B();
```



```
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```

Note : If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

4. **Subpackage :** Package inside the package is called the ***subpackage***. It should be created to categorize the package further.

EXAMPLE

```
//The below code illustrate creation of package
//save by A.java
package p1;
public class A
{
public void msg()
 {
System.out.println("MY NAME IS ANIL");
 }
public static void main(String args[])
 {
   A obj=new A();
obj.msg();
 }
}
//The below code illustrate creation of sub-package
//save by A1.java
package p1.subpack;
public class A1
{
```

```
public void msg1()
 {
System.out.println("I HAVE PASSED TENTH CLASS");
 }
public static void main(String args[])
 {
   A1 obj1=new A1();
obj1.msg1();
 }
}
//save by B.java
package p2;
class B
{
void display()
 {
System.out.println("I AM STUDYING DCME FINAL YEAR");
 }
public static void main(String args[])
 {
p1.Aobj=new p1.A();
obj.msg();
    B obj2=new B();
obj2.display();
 }
}
```

Output :

```
MY NAME IS ANIL
I AM STUDYING DCME FINAL YEAR
```


3.7 EXPLORING IO AND UTIL PACKAGES

The I/O classes defined by java.io are listed in the below table.

BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

The interfaces defined by java.io are listed below :

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

JAVA.UTIL PACKAGE

It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Following are the Important Classes in Java.utilpackage :

1. **Abstract Collection :** This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
2. **Abstract List :** This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

3. **AbstractMap<K,V>** : This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
4. **AbstractMap.SimpleEntry<K,V>** : An Entry maintaining a key and a value.
5. **AbstractMap.SimpleImmutableEntry<K,V>** : An Entry maintaining an immutable key and value.
6. **AbstractQueue** : This class provides skeletal implementations of some Queue operations.
7. **AbstractSequentialList** : This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
8. **AbstractSet** : This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.
9. **ArrayDeque** : Resizable-array implementation of the Deque interface.
10. **ArrayList** : Resizable-array implementation of the List interface.
11. **Arrays** : This class contains various methods for manipulating arrays (such as sorting and searching).
12. **BitSet** : This class implements a vector of bits that grows as needed.
13. **Calendar** : The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
14. **Collections** : This class consists exclusively of static methods that operate on (or return collections.
15. **Currency** : Represents a currency.
16. **Date** : The class Date represents a specific instant in time, with millisecond precision.
17. **Dictionary<K,V>** : The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
18. **EnumMap,V>** : A specialized Map implementation for use with enum type keys.
19. **EnumSet** : A specialized Set implementation for use with enum types.
20. **EventListenerProxy** : An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
21. **EventObject** : The root class from which all event state objects shall be derived.

22. **FormattableFlags** : FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables.
23. **Formatter** : An interpreter for printf-style format strings.
24. **GregorianCalendar** : GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
25. **HashMap<K,V>** : Hash table based implementation of the Map interface.
26. **HashSet** : This class implements the Set interface, backed by a hash table (actually a HashMap instance).
27. **Hashtable<K,V>** : This class implements a hash table, which maps keys to values.
28. **IdentityHashMap<K,V>** : This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
29. **LinkedHashMap<K,V>** : Hash table and linked list implementation of the Map interface, with predictable iteration order.
30. **LinkedHashSet** : Hash table and linked list implementation of the Set interface, with predictable iteration order.
31. **LinkedList** : Doubly-linked list implementation of the List and Deque interfaces.
32. **PriorityQueue** : An unbounded priority queue based on a priority heap.
33. **Random** : An instance of this class is used to generate a stream of pseudorandom numbers.
34. **ResourceBundle** : Resource bundles contain locale-specific objects.
35. **Scanner** : A simple text scanner which can parse primitive types and strings using regular expressions.
36. **ServiceLoader** : A simple service-provider loading facility.
37. **SimpleTimeZone** : SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
38. **Stack** : The Stack class represents a last-in-first-out (LIFO) stack of objects.
39. **StringTokenizer** : The string tokenizer class allows an application to break a string into tokens.
40. **Timer** : A facility for threads to schedule tasks for future execution in a background thread.
41. **TimerTask** : A task that can be scheduled for one-time (or) repeated execution by a Timer.

3.8 VARIOUS STREAM CLASSES

Streams in Java represent an ordered sequence of data. Java performs input and output operations in the terms of streams.

It uses the concept of streams to make I/O operations fast. For example, when we read a sequence of bytes from a binary file, actually, were reading from an input stream.

Similarly, when we write a sequence of bytes to a binary file, were writing to an output stream.

Java supports two types of I/O streams. They are :

- Byte streams.
- Character streams.
- Byte Streams.

Byte streams in Java are designed to provide a convenient way for handling the input and output of bytes. We use them for reading (or) writing to binary data I/O.

Byte streams are especially used when we are working with binary files such as executable files, image files, and files in low-level file formats such as .zip, .class, .obj, and .exe.

Byte streams that are used for reading are called ***input streams*** and for writing are called ***output streams***. They are represented by the abstract classes of InputStream and OutputStream in Java.

1. **Character Streams : Character streams in Java** are designed for handling the input and output of characters. They use 16-bit Unicode characters. Character streams are more efficient than byte streams. They are mainly used for reading (or) writing to character (or) text-based I/O such as text files, text documents, XML, and HTML files.

   Text files are those files that are human readable. For example, a .txt file that contains human-readable text. This file is created with a text editor such as Notepad in Windows.

   Character streams that are used for reading are called ***readers*** and for writing are called ***writers***. They are represented by the abstract classes of Reader and Writer in Java.

2. **Stream Classes :** All streams in Java are represented by classes in java.io package. This package contains a lot of stream classes that provide abilities for processing all types of data.

S.No.	InputStream	Description
1.	BufferedInputStream	It adds buffering abilities to an input stream. In simple words, it buffered input stream.
2.	ByteArrayInputStream	Input stream that reads data from a byte array.
3.	DataInputStream	It reads bytes from the input stream and converts them into appropriate primitive-type values (or) strings.
4.	FileInputStream	This input stream reads bytes from a file.
5.	FilterInputStream	It filters bytes from an input stream.
6.	ObjectInputStream	Input stream for objects.
7.	PipedInputStream	It creates a communication channel on which data can be received.
8.	PushbackInputStream	It is a subclass of FilterInputStream that adds "pushback" functionality to an input stream.
9.	SequenceInputStream	It is used to read input streams sequentially, one after the other.

INPUTSTREAM METHODS

The abstract InputStream class in Java defines several methods for performing input functions such as reading bytes, closing streams, marking positions in streams, etc.

All these methods are present in java.io.InputStream package. InputStream methods are as follows :

(i) ***int Read() :*** The read() method reads the next byte of data from the input stream. It returns input byte read as an int value in the range 0 to 255.

If no byte is present because the end of the stream is reached, the value 1 is returned. If this method encounters an I/O error, it will throw an IOException.

The read() method returns an int value -1 instead of a byte because it indicates the end of stream.

(ii) ***int read(byte[ ] b) :*** This method reads the number of bytes from the input stream and stores them into the array of bytes b.

It returns the total number of bytes read as an int. If the end of stream is reached, returns -1.

(iii) ***int read(byte[ ] b, int n, int m) :*** It reads up to m bytes of data from the input stream starting from nth byte into an array b.

It returns the total number of bytes read as an int. Returns -1 at the end of stream because of no more data.

(iv) ***int available() :*** The available method returns an estimate of the number of bytes that can be read (or) skipped over from the input stream.

(v) ***void close() :*** The close() method closes the input stream and releases any system resources associated with it.

(vi) ***long skip(long n) :*** This method skips over n bytes of data from this input stream. It returns the actual number of bytes skipped.

(vii) ***void reset() :*** The reset() method is used to go back to the beginning of the stream.

(viii) ***booleanmarkSupported() :*** This method tests this input stream supports the mark and reset methods. It returns true if the input stream supports mark and reset methods.

2. **OutputStream Classes :** OutputStream class is an abstract class. It is the root class for writing binary data. It is a superclass of all classes that represents an output stream of bytes. we cannot create object with using subclass

Since like InputStream, OutputStream is an abstract class, therefore, we cannot create object of it. The hierarchy of classification of OutputStream classes has shown in the above diagram.

The several subclasses of OutputStream class in Java can be used for performing several output functions. They are listed with a brief description in the below table.

S.No.	OutputStream Subclass	Description
1.	ByteArrayOutputStream	Output stream that writes data to a byte array.
2.	DataOutputStream	It converts primitive-type values (or) strings into bytes and outputs bytes to the stream.
3.	FileOutputStream	It writes byte stream into a file.
4.	FilterOutputStream	It filters bytes from an output stream.
5.	ObjectOutputStream	Output stream for objects.
6.	PipedOutputStream	It creates a communication channel on which data can be sent.

OUTPUTSTREAM METHODS

The OutputStream class defines the following method in java.io.OutputStream package that are used to perform output tasks such as writing bytes, closing streams, and flushing streams.

The OutputStream methods with a brief description are as follows :

(i) ***void write(int b)*** : The write() method writes the specified byte to the output stream. It accepts an int value as an input parameter. It throws an IOException if an I/O error occurs (Eg : output stream has been closed).

(ii) ***void write(byte[ ] b)*** : This method writes all the specified bytes in the array b to the output stream.

(iii) ***void write(byte[ ] b, int n, int m)*** : It writes m bytes from array b starting from nth byte to the output stream.

(iv) ***void close()*** : It closes the output stream and releases any system resources associated with this stream.

(v) ***void flush()*** : It flushes the output stream and forces any buffered output bytes to be written.

CharacterStream classes in Java are used to read and write 16-bit Unicode characters.

In other words, character stream classes are mainly used to read characters from the source and write them to the destination.

They can perform operations based on characters, char arrays, and Strings.

Like byte stream classes, character stream classes also contain two kinds of classes. **They are named as :**

- Reader Stream classes.
- Writer Stream classes.

Lets understand in detail reader stream classes and writer stream classes one by one.

1. **Reader Stream Classes :** Reader stream classes are used for reading characters from files.

The hierarchy of Reader Stream Classes is shown in the Fig. 3.3.

The classes belonging to Reader stream classes are very similar functionality to Input Stream classes. The only difference is that Input Stream uses bytes, whereas Reader Stream classes use characters.

In fact, both byte and character stream classes use the same methods. Therefore, reader classes can perform all the operations implemented by input stream classes.

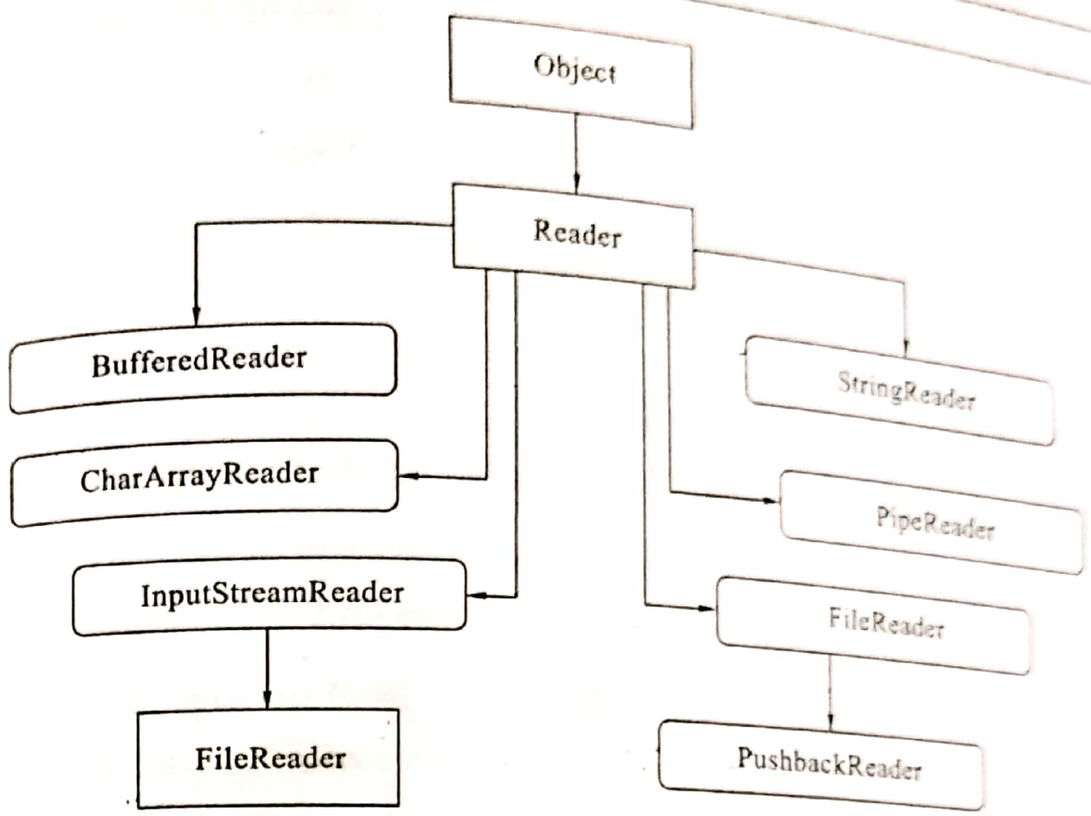


FIG 3.3 : Hierarchy of Reader Stream Classes

CONSTRUCTOR OF READER CLASS

Reader class in Java defines the following constructors with protected access modifiers. They are :

1. **protected Reader() :** This form of constructor creates a new character-stream reader whose critical sections will synchronize on the reader itself.
2. **protected Reader(Object lock) :** This form of constructor creates a new character-stream reader whose critical sections will synchronize on the specified object.

READER SUBCLASSES

Lets understand the subclasses of the Reader class in a brief description.

1. **BufferedReader :** This class is used to read characters from the buffered input character stream.
2. **CharArrayReader :** This class is used to read characters from the char array (or) character array.
3. **FileReader :** This class is used to read characters (or contents) from a file.
4. **FilterReader :** This class is used to read characters from the underlying character input stream.
5. **InputStreamReader :** This class is used to translate (or) convert bytes to characters.

6. **PipeReader** : This class is used to read characters from the connected piped output stream.
7. **StringReader** : This class is used to read characters from a string.
8. **PushBackReader** : This class allows one (or) more characters to be returned to the input stream.

READER CLASS METHODS

The Reader class methods defined the following methods that are as follows :

1. **int read()** : This method returns an integer representation of the next character present in the invoking input stream. It returns -1 when the end of the input is encountered.
2. **int read(char buffer[ ])** : This method is used to read up to buffer.length characters from the specified buffer. It returns the actual number of characters successfully read. It returns -1 when the end of the input is encountered.
3. **int read(char buffer[ ], intloc, intnumChars)** : This method is used to read up to numChars characters from the buffer starting at the specified location. It returns the number of characters successfully read. 1 is returned when the end of the input is encountered.
4. **void mark(intnumChars)** : This method is used to mark the current point in the input stream that will remain until numChars characters are read.
5. **void reset()** : The reset() method is used to reset the input pointer to the preceding set mark.
6. **long skip(long numChars)** : The skip() method is used to skip the specified numChars characters from the input stream and returns the number of characters actually skipped.
7. **boolean ready()** : This method returns a true value if the next request of input is ready (i.e. not wait). Otherwise, it returns false.
8. **void close()** : This method is used to close the input stream. If the program attempts to read the input further, it generates IOException.
9. **booleanmarkSupported( )** : This method returns boolean value true if mark( )/reset( ) are supported on this stream.

All the methods provided by the Reader class (except for markSupported( )) will throw an IOException on error conditions.

WRITER STREAM CLASSES

Writer stream classes are used to write characters to a file. In other words, They are used to perform all output operations on files.

Writer stream classes are similar to output stream classes with only one difference that output stream classes use bytes to write while writer stream classes use characters to write.

The hierarchy diagram of subclasses of the Writer class is shown in the below Fig. 3.4

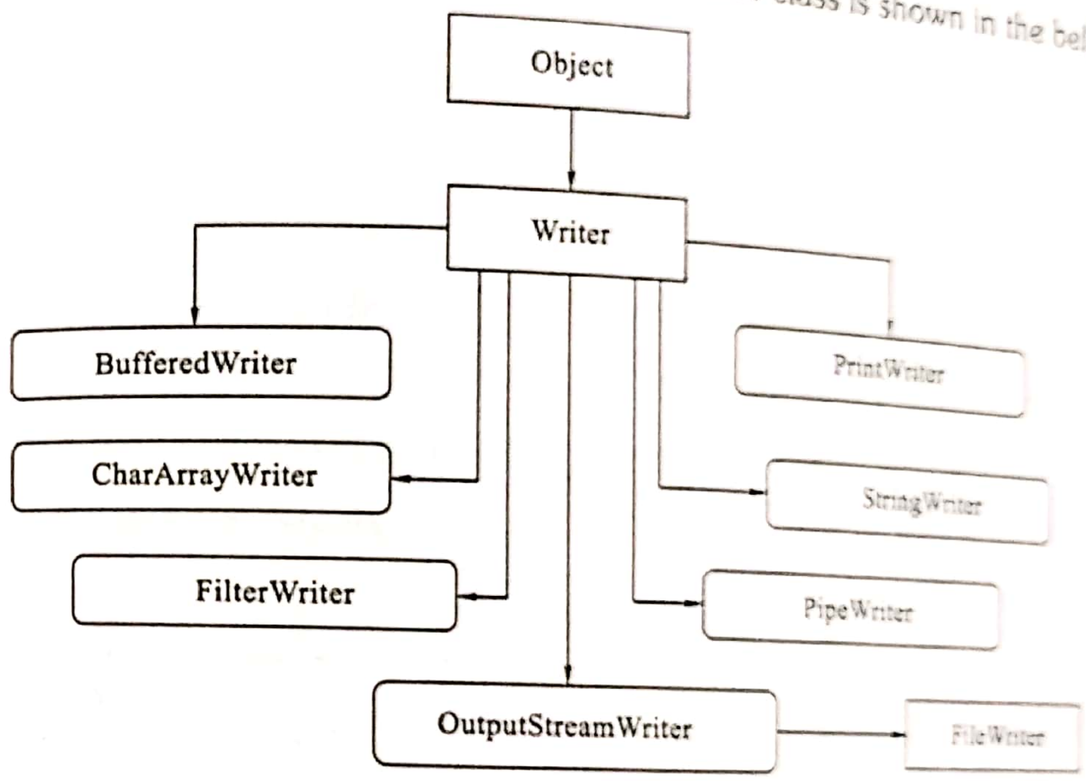


FIG 3.4 : Hierarchy of Writer Stream Classes

CONSTRUCTOR OF WRITER CLASS

Writer class in Java defines the following constructors with protected access modifiers that are as follows :

1. **protected Writer() :** This form of constructor constructs a new character-stream writer whose critical sections will synchronize on the writer itself.
2. **protected Writer(Object lock) :** This form of constructor constructs a new character-stream writer whose critical sections will synchronize on the given object.

WRITER SUBCLASSES

Lets understand the subclasses of writer class in a brief description.

1. **BufferedWriter :** This class is used to write characters to the buffered output character stream.

2. **FileWriter** : This output stream class writes characters to the file.
3. **CharArrayWriter** : This output stream class writes the characters to the character array
4. **OutpuStreamWriter** : This output stream class translates (or) converts from bytes to characters.
5. **PipedWriter** : This class writes the characters to the piped output stream.
6. **StringWriter** : This output stream class writes the characters to the string.
7. **PrintWriter** : This output stream class contains print() and println().
8. **FilterReader** : This class is used to write characters on the underlying character output stream.

WRITER CLASS METHODS

To write the characters to the output stream, Writer stream class in Java defines the following methods that are as follows :

1. **void write()** : The write() method is used to write the data to the invoking output stream.
2. **void write(intch)** : This method is used to write a single character to the invoking output stream.
3. **void write(char buffer[ ])** : This method is used to write a complete array of characters to the invoking output stream.
4. **void write(char buffer [ ], intloc, intnumChars)** : This method is used to write a subrange of numChars characters from the character array, starting at the specified location to the output stream.
5. **void write(String str)** : This method is used to write str to the invoking output stream.
6. **void write(String str, intloc, intnumChars)** : This method writes a subrange of numChars characters from the string str, starting at the specified location.
7. **void close ()** : This method is used to close the output stream. It will produce an IOException if an attempt is made to write to the output stream after closing the stream.
8. **void flush()** : This method flushes the output stream and writes the waiting buffered characters.
9. **Writer append(char ch)** : The append() method appends character ch to the end of the invoking output stream. It returns a reference to the invoking output stream.
10. **Writer append(CharSequence chars)** : This method appends chars to the end of the invoking output stream. It returns a reference to the invoking output.

REVIEW QUESTIONS

1 Mark Questions

1. Define a package.
2. List different types of packages.
3. Define built-in package
4. Define user-defined package
5. List different access specifiers in java.
6. Write the syntax to create a package.
7. Define a subpackage.
8. What is astream.
9. List the two types of stream classes.
10. List the two types of byte stream classes.
11. List the two types of character stream classes.

3 Marks Questions

1. Write the advantages of packages.
2. Write about different types of packages.
3. Write the use of java.lang, java.io and java.util packages.
4. Write about different types of access specifiers.
5. Write a java program to create a user defined package.
6. List input stream classes in java.
7. List output stream classes in java.
8. Write a java program to import packages using fully qualified name.
9. Write a java program to import packages using packagename.classname.
10. Write short notes on byte stream and characterstream.

5 Marks Questions

1. Explain about Java API packages.
2. Explain the concept of access protections.

3. Explain about access specifiers with example programs.
4. Explain the concept of creating user-defined package with an example program.
5. Explain the concept of importing packages in java with example
6. Write a java program to import packages using fully qualified name and packagename.classname.
7. Explain about java.io packages.
8. Explain about java.util packages.
9. Explain about various stream classes.
10. Explain about sub-packages with an example program.

CHAPTER 4

CONCEPTS OF APPLETS, AWT AND EVENT HANDLING

CHAPTER OUTLINE

4.1 APPLET AND LIFE CYCLE OF AN APPLET

In the real time, Applications are classified into two types. They are :

- Stand-alone applications.
- Distributed applications.

STANDALONE APPLICATIONS

Standalone application is one which runs in the context of local disk and whose results are not shareable throughout the world. standalone application contains main() for executing a java program and System.out.println() to display results on moniter.

DISTRIBUTED APPLICATIONS

Distributed application is one which runs in the context of browser (internet explorer, Netscape navigator, opera mini, fire fox, chrome)and whose results are sharable across world.

Distributed application contains life cycle methods to execute a java program and some predefined methods to display results on browser.

DEFINITION OF AN APPLET

An applet is a Java program that runs in a Web browser.Applets are designed to be embedded within an HTML page.

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

There are two types of applets. They are :

- Local applet.
- Remote applet.
- Advantages of Applets.

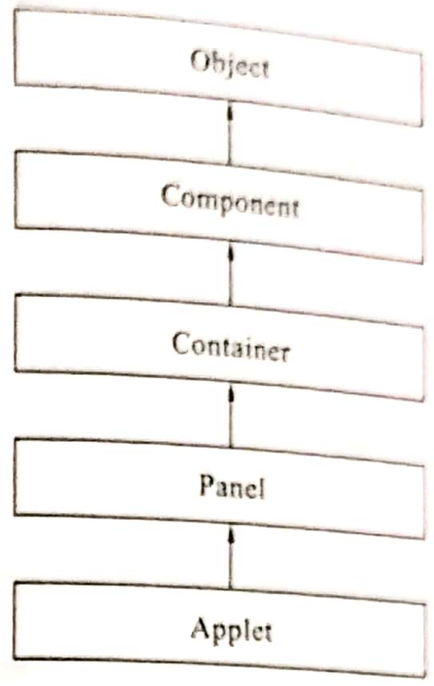
There are many advantages of applet. They are as follows :

- It works at client side so less response time.
- Secured.
- It can be executed by browsers running under many platforms, including Linux, Windows, MacOs etc.,

DRAWBACK OF APPLET

Plugging is required at client browser to execute applet.

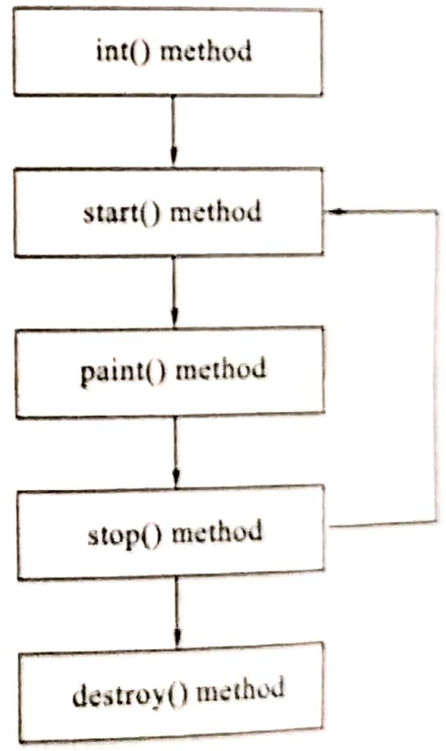
HIERARCHY OF APPLETS



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component. Component is a subclass of Object class.

LIFE CYCLE METHODS

An applet undergoes various states between its object creation and object removal is known as ***Applet Life Cycle.*** Each state is represented by a method. There exist five states represented by five methods. Life cycle of an applet is given below Fig.



The java.applet.Applet class contains four life cycle methods and java.awt.Component class provides one method for executing an applet.

Every java applet program supports following life cycle methods.

1. **public void init() :** This method is called by browser when user makes first request.

   This method is called by browser only once. It is used to initialize the Applet. It is invoked only once.

   This method is used to perform one time operations.

   Examples of one time operations.

   - Opening a file.
   - Getting a database connection.

2. **public void start() :**

   This method is called by the browser each and every time to perform repeated operations.

   It is invoked after the init() method (or) when browser is maximized. It is used to start the Applet.

   Examples :

   - Reading a record from file.
   - Reading a record from database.

3. **public void stop() :** It is used to stop the Applet. It is invoked when Applet is stopped (or) browser is minimized.

   This method is called by browser when user minimizes a window.

4. **public void destroy() :** It is used to destroy the Applet. It is invoked only once.

   This method is called by browser when the user closes a window.

   The Component class provides one method of applet.

5. **public void paint() :** This method is used to display results on a browser. It is used to paint the Applet.

   paint() is not a life cycle method.

Note : Init() is used to obtain resources like file and database connection and destroy() is used to close the resources which are obtained by init().

4.2 CREATION OF APPLETS WITH EXAMPLE PROGRAMS

APPLET SKELETON CODE

It contains five methods in which four methods- init(), start(), stop() ,destroy() are defined by Applet and paint() method is defined by AWT component class.


```
import java.applet.* ;
import java.awt.* ;
/*
<applet code="appletskel" width=200 height=200>
</applet>
*/
public class appletskel extends Applet
{
public void init()
{
initialization
}
public void start()
{
start or resume execution
}
public void stop()
{
suspend execution
}
public void destroy()
{
terminating applet
}
public void paint(Graphics g)
{
   Display the contents of window
}
}
```

There are two ways to run an applet :

1. By html file.
2. By appletViewer tool.

1. **Simple Example of Applet by html File :** To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

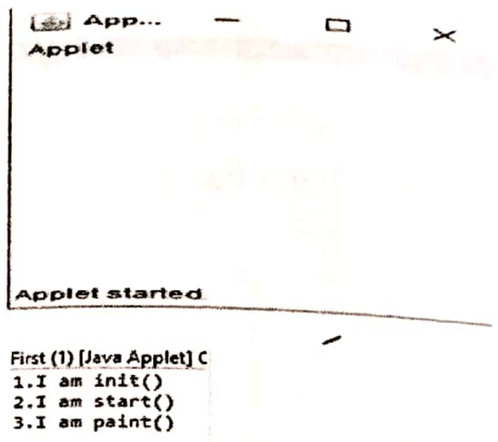
```
//AppletLifeCycle.java
import java.applet.Applet;
import java.awt.Graphics;
public class AppletLifeCycle extends Applet
{
public void init()
{
System.out.println("I am init()");
}
public void start()
{
System.out.println("I am start()");
}
public void paint(Graphics g)
{
System.out.println("I am paint()");
}
public void stop()
{
System.out.println("I am stop()");
}
public void destroy()
{
System.out.println("I am destroy()");
}
}
```

Note : class must be public because its object is created by Java Plugin software that resides on the browser.

HTML Code: AppletLifeCycle.html

```
<!DOCTYPE/html>
<html>
<body>
<applet code="AppletLifeCycle.class" width="300" height="300"></applet>
</body>
</html>
```

Output :



After **minimizing the applet-Output :**

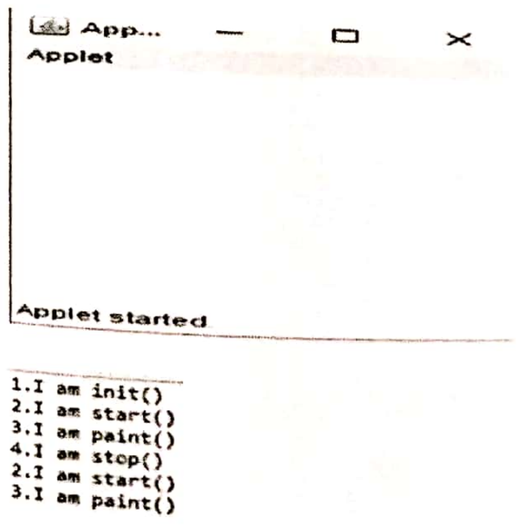
I am init()

I am start()

I am paint()

I am stop()

After **maximizing the applet-Output :**



After closing the applet-Output :

I am init()

I am start()

I am paint()

I am stop()

I am start()

I am paint()

I am stop()

I am destroy()

2. **Simple Example of Applet by Apple tviewer Tool :** To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required.

```
//AppletLifeCycle.java
import java.applet.Applet;
import java.awt.Graphics;
public class AppletLifeCycle extends Applet
{
public void init()
{
System.out.println("I am init()");
}
public void start()
{
System.out.println("I am start()");
}
public void paint(Graphics g)
{
System.out.println("I am paint()");
}
public void stop()
{
System.out.println("I am stop()");
}
```



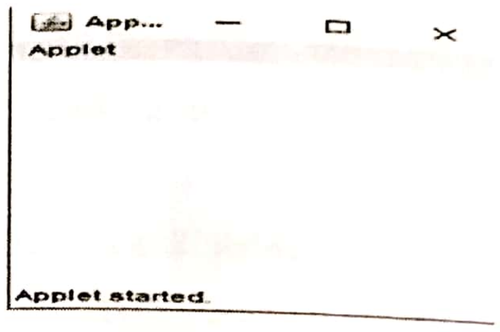
```
public void destroy()
{
System.out.println("I am destroy()");
}
}
/*<applet code="AppletLifeCycle.class" width="300" height="300"></applet>
*/
```

Output :

To execute the applet by appletviewer tool, write in command prompt

c:\>javac AppletLifeCycle.java

c:\>appletviewer AppletLifeCycle.java




```
First (1) [Java Applet] C
1.I am init()
2.I am start()
3.I am paint()
```

After minimizing the applet-Output :

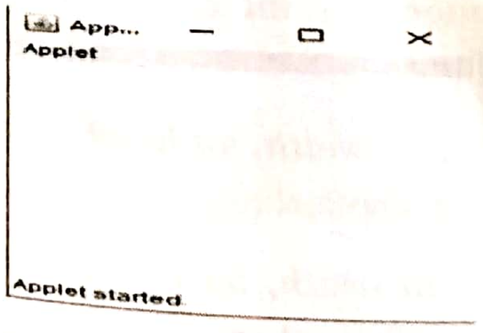
I am init()

I am start()

I am paint()

I am stop()

After maximizing the applet-Output :




```
1.I am init()
2.I am start()
3.I am paint()
4.I am stop()
2.I am start()
3.I am paint()
```

After closing the applet-Output :

```
I am init()
I am start()
I am paint()
I am stop()
I am start()
I am paint()
I am stop()
I am destroy()
```

3. **Displaying Graphics in Applet :** java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class :

(i) ***public abstract void drawString(String str, int x, int y) :*** It is used to draw the specified string.

(ii) ***public void drawRect(int x, int y, int width, int height) :*** Draws a rectangle with the specified width and height.

(iii) ***public abstract void fillRect(int x, int y, int width, int height) :*** It is used to fill rectangle with the default color and specified width and height.

(iv) ***public abstract void drawOval(int x, int y, int width, int height) :*** It is used to draw oval with the specified width and height.

(v) ***public abstract void fillOval(int x, int y, int width, int height) :*** It is used to fill oval with the default color and specified width and height.

(vi) ***public abstract void drawLine(int x1, int y1, int x^2, int y^2) :*** It is used to draw line between the points(x^1, y^1) and (x^2, y^2).

(vii) ***public abstract booleandrawImage(Image img, int x, int y, ImageObserver observer) :*** It is used draw the specified image.

(viii) ***public abstract void drawArc(int x, int y, int width, int height, intstartAngle, intarcAngle) :*** It is used draw a circular or elliptical arc.

(ix) ***public abstract void fillArc(int x, int y, int width, int height, intstartAngle, intarcAngle) :*** It is used to fill a circular or elliptical arc.

(x) ***public abstract void setColor(Color c) :*** It is used to set the graphics current color to the specified color.

(xi) **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

EXAMPLE

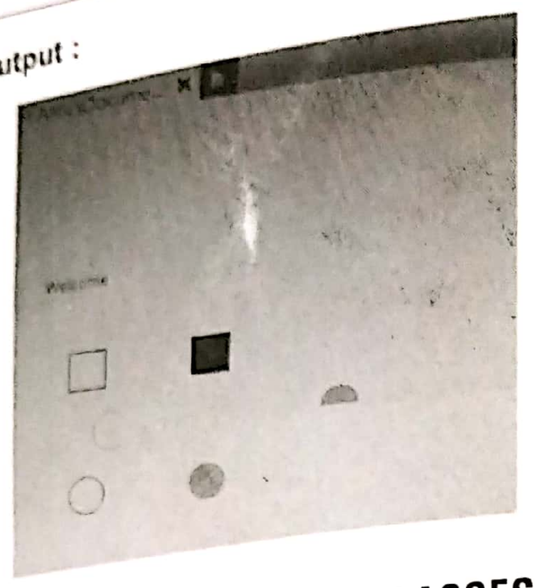
Write a java applet program to illustrate Graphics.

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet
{
public void paint(Graphics g)
{
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Output :



4.3 LIST AND DISCUSS AWT CLASSES

AWT stands for Abstract window toolkit is an Application programming interface (API) for creating Graphical User Interface (GUI) in Java.It allows Java programmers to develop window-based applications.

The AWT API in Java consists of a comprehensive set of classes and methods that are required for creating and managing the Graphical User Interface (GUI) in a simplified manner.

FEATURES OF AWT

- AWT provides Graphics and imaging tools, such as shape, color and font classes.
- AWT also avails layout managers which helps in increasing the flexibility of the window layouts.
- Data transfer classes are also a part of AWT that helps in cut-and-paste through the native platform clipboard.
- It supports a wide range of libraries that are necessary for creating graphics for gaming products, banking services, educational purposes, etc.,
- AWT contains a set of native user interface components.
- It supports a robust event-handling model.

AWT HIERARCHY

AWT provides various components like button, label, checkbox, etc., used as objects inside a Java Program. AWT components use the resources of the operating system, i.e., they are platform-dependent, which means, component's view can be changed

according to the view of the operating system. The classes for AWT are provided by the Java.awt package for various AWT components. The hierarchy of Java AWT classes are given below.

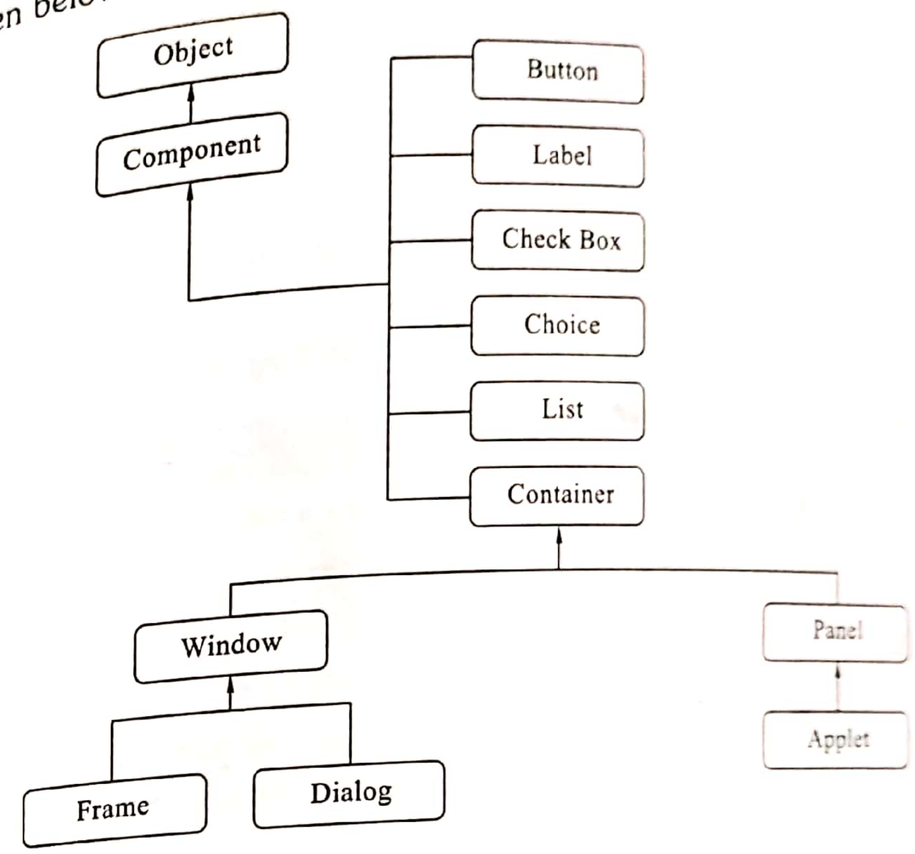


FIG 4.1 :

COMPONENTS

All the elements like the button, text fields, scroll bars, etc. are called *components*. In Java AWT, In order to place every component in a particular position on a screen, we need to add them to a container.

CONTAINER

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

It is basically a screen where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Types of Containers :

There are four types of containers in Java AWT :

1. Window
2. Panel
3. Frame
4. Dialog

1. **Window :** The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

2. **Panel :** The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

3. **Frame :** The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc., Frame is most widely used container while developing an AWT application.

S.No.	Method	Description
1.	public void add(Component c)	Inserts a component on this component.
2.	public void setSize(intwidth,int height)	Sets the size (width and height) of the component.
3.	public void setLayout(LayoutManager m)	Defines the layout manager for the component.
4.	public void setVisible(boolean status)	Changes the visibility of the component, by default false.

4.4 AWT CONTROLS WITH EXAMPLE PROGRAMS

Java supports various types of AWT controls. They are given below.

LABEL

The easiest control to use is a label.

Labels are passive controls that do not support any interaction with the user.

Label defines the following constructors :

Label( ) : It creates a blank label.

Label(String str) : It creates a label that contains the string specified by str.

Label(String str, int how) : It creates a label that contains the string specified by str using the alignment specified by how.

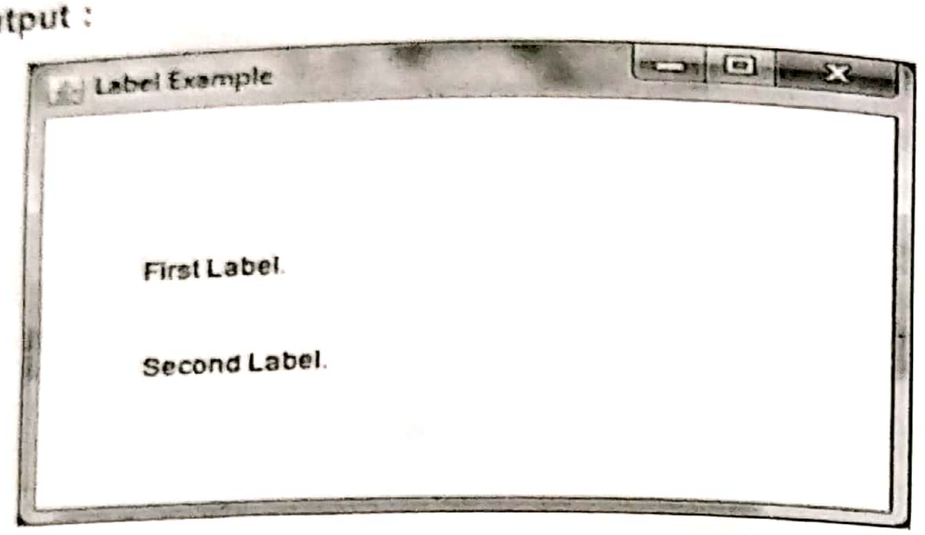
The alignment is specified by one of these three constants.

Label.LEFT

Label.RIGHT

Label.CENTER

Output :



BUTTON

The most widely used control is the button.

A push button is a component that contains a label and that generates an event when it is pressed.

Button defines these two constructors :

- **Button( ) :** It creates an empty button
- **Button(String str) :** It creates a button that contains str as a label

After a button has been created, you can set its label by calling setLabel( )

voidsetLabel(String str)

Here, str becomes the new label for the button.

EXAMPLE :

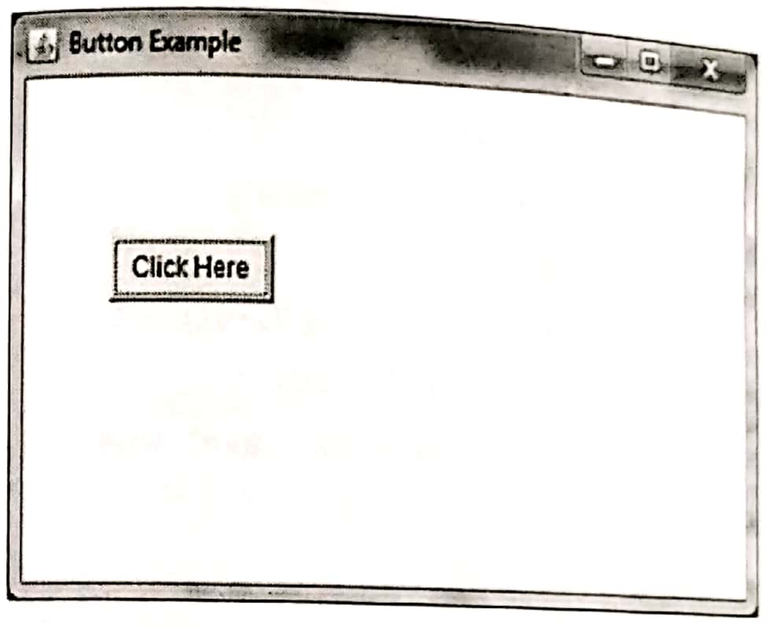
Write a java program to illustrate Button.

```
import java.awt.*;
public class ButtonExample
{
public static void main (String[] args)
{
  // create instance of frame with the label
  Frame f = new Frame("Button Example");
  // create instance of button with label
  Button b = new Button("Click Here");
```



```
  // set the position for the button in frame
  b.setBounds(50,100,80,30);
  // add button to the frame
  f.add(b);
  // set size, layout and visibility of frame
  f.setSize(400,400);
  f.setLayout(null);
  f.setVisible(true);
}
}
```

Output :



CHECKBOX

A check box is a control that is used to turn an option on (or) off.

It consists of a small box that can either contain a check mark (or) not.

There is a label associated with each check box that describes what option the box represents.

You change the state of a check box by clicking on it.

Check boxes are objects of the Checkbox class.

Checkbox supports these constructors :

Checkbox( ) : It creates a check box whose label is initially blank. The state of the check box is unchecked.

Checkbox(String str) : It creates a check box whose label is specified by str. The state of the check box is unchecked.

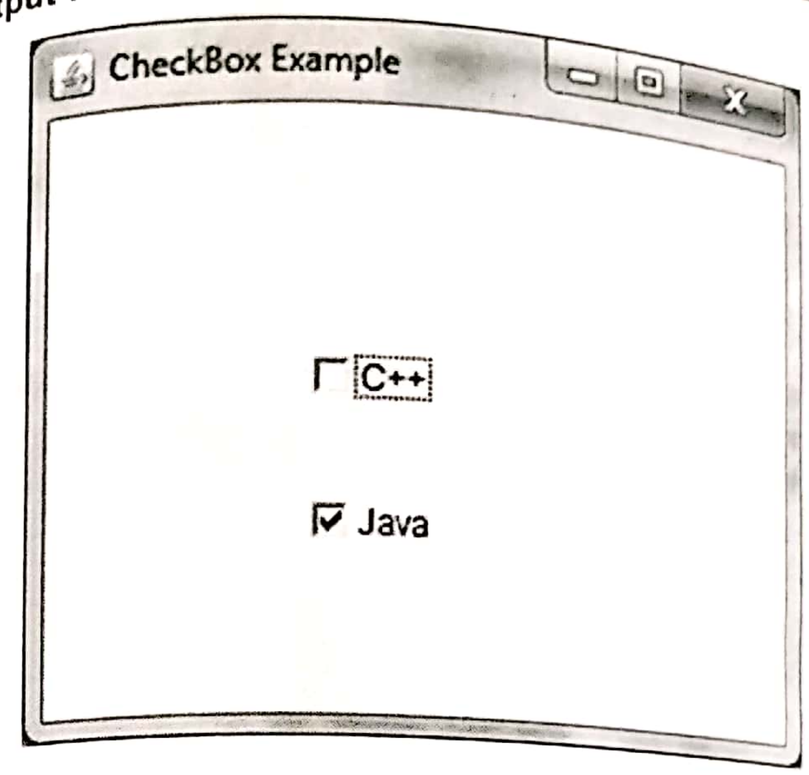
Checkbox(String str, boolean on) : It allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared.

EXAMPLE

Write a java program to illustrate Checkbox.

```
import java.awt.*;
public class CheckboxExample1
{
// constructor to initialize
    CheckboxExample1()
  {
    // creating the frame with the title
     Frame f = new Frame("Checkbox Example");
     // creating the checkboxes
      Checkbox checkbox1 = new Checkbox("C++");
      checkbox1.setBounds(100, 100,  50, 50);
       Checkbox checkbox2 = new Checkbox("Java", true);
     // setting location of checkbox in frame
 checkbox2.setBounds(100, 150,  50, 50);
     // adding checkboxes to frame
      f.add(checkbox1);
      f.add(checkbox2);
     // setting size, layout and visibility of frame
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main (String args[])
  {
    new CheckboxExample1();
  }
}
```


Output :



LIST

The List class provides a compact, multiple-choice, scrolling selection list. It can also be created to allow multiple selections.

List provides these constructors

List( ) : List control that allows only one item to be selected at any one time.

List(int numRows) : the value of numRows specifies the number of entries in the list that will always be visible .

List(int numRows, boolean multipleSelect) : if multipleSelect is true, then the user may select two or more items at a time.

To add a selection to the list, we have to call add( )

It has the following two forms

- **void add(String name)** : adds items to the end of the list.
- **void add(String name, int index)** : adds the item at the index specified by index

Here, name is the name of the item added to the list

For lists that allow only single selection you can determine which item is currently selected by calling following methods.

String getSelectedItem( )

int getSelectedIndex( )

For lists that allow multiple selection you must use following methods.

String[ ] getSelectedItems( )

int[ ] getSelectedIndexes( )

EXAMPLE

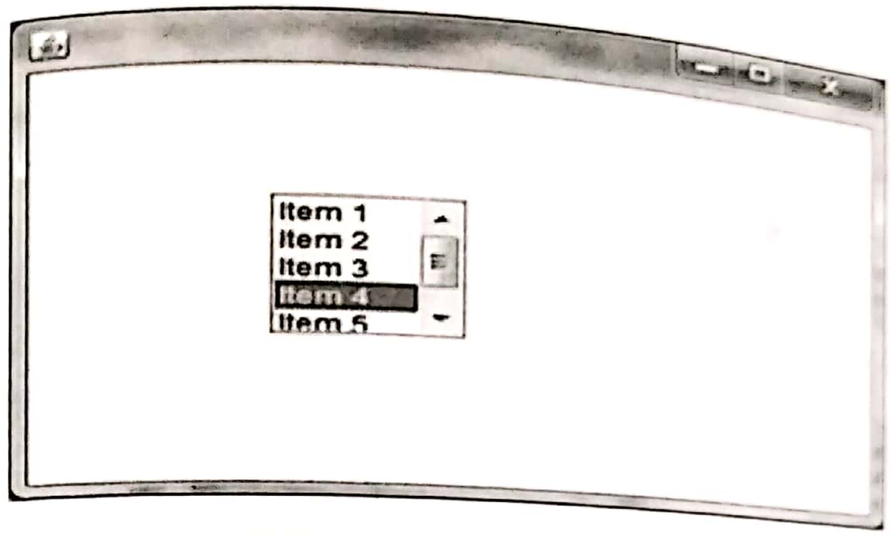
Write a java program to illustrate List.

```
import java.awt.*;
public class ListExample1
{
   // class constructor
    ListExample1()
   {
    // creating the frame
     Frame f = new Frame();
   // creating the list of 5 rows
    List l1 = new List(5);
     // setting the position of list component
   l1.setBounds(100, 100, 75, 75);
     // adding list items into the list
   l1.add("Item 1");
   l1.add("Item 2");
   l1.add("Item 3");
   l1.add("Item 4");
   l1.add("Item 5");
   // adding the list to frame
   f.add(l1);
   // setting size, layout and visibility of frame
   f.setSize(400, 400);
   f.setLayout(null);
   f.setVisible(true);
```



```
   }
  public static void main(String args[])
  {
   new ListExample1();
  }
}
```

Output :



SCROLL BARS

Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Scroll bars are used to select continuous values between a specified minimum and maximum values.

Scroll bars may be oriented horizontally or vertically.

Scroll bars are encapsulated by the Scrollbar class.

Scrollbar defines the following constructors :

- **Scrollbar( )** : It creates a vertical scroll bar.
- **Scrollbar(int style)** : It allow you to specify the orientation of the scroll bar.

If style is Scrollbar.VERTICAL, then a vertical scroll bar is created.

If style is Scrollbar.HORIZONTAL, then a horizontal scroll bar is created.

Scrollbar(int style, int initialValue, int thumbSize, int min, int max) : In the this form of the constructor, the initial value of the scroll bar is passed in initialValue

The number of units represented by the height of the thumb is passed in thumbSize

The minimum and maximum values for the scroll bar are specified by min and max setValues( )is used to change the parameters.

The syntax of it is

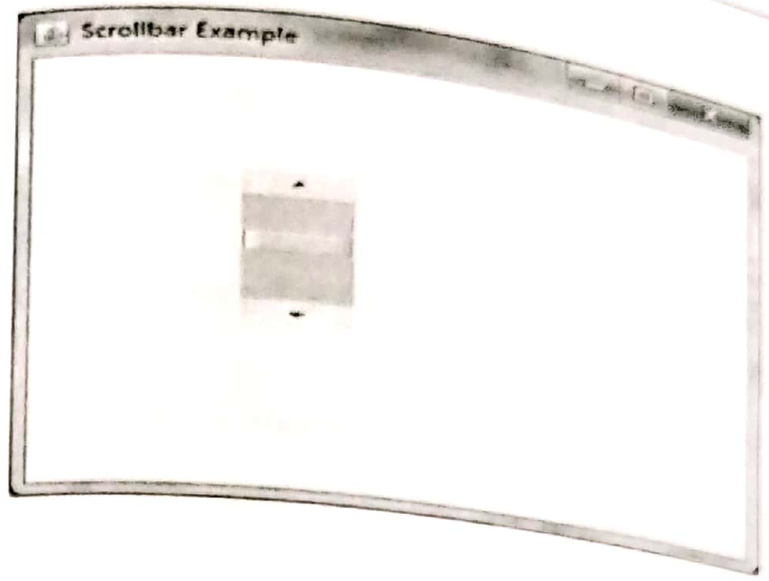
```
void setValues(int initialValue, int thumbSize, int min, int max)
```

EXAMPLE

Write a java program to illustrate Scrollbar.

```
// importing awt package
import java.awt.*;
 public class ScrollbarExample1
{
ScrollbarExample1()
{
    // creating a frame
        Frame f = new Frame("Scrollbar Example");
    // creating a scroll bar
        Scrollbar s = new Scrollbar();
    // setting the position of scroll bar
        s.setBounds (100, 100, 50, 100);
    // adding scroll bar to the frame
        f.add(s);
    // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
    new ScrollbarExample1();
}
}
```


Output :



TEXT FIELD

The TextField class implements a single-line text-entry area.

Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

TextField is a subclass of TextComponent.

TextField defines the following constructors :

- **TextField( ) :** It creates a default text field.
- **TextField(int columns) :** It creates a text field with a specified number of columns.
- **TextField(String str) :** It creates a text field initialized with the string contained in str.

TextField(String str, int numChars) : It creates a text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

- To obtain the string currently contained in the text field, we have to use String getText( ).
- To set the text, we have to use void setText(String str).

EXAMPLE

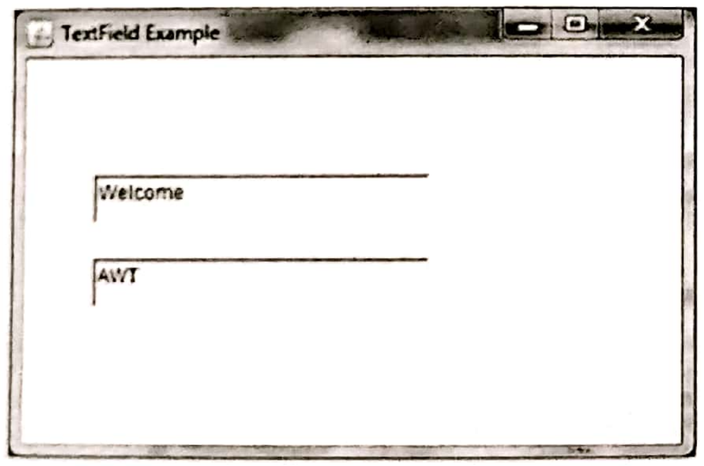
Write a java program to illustrate TextField.

```
// importing AWT class
import java.awt.*;
public class TextFieldExample1
{

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

```
// creating a frame
Frame f = new Frame("TextField Example");
// creating objects of textfield
TextField t1, t2;
// instantiating the textfield objects
// setting the location of those objects in the frame
t1 = new TextField("Welcome");
t1.setBounds(50, 100, 200, 30);
t2 = new TextField("AWT");
t2.setBounds(50, 150, 200, 30);
// adding the components to frame
f.add(t1);
f.add(t2);
// setting size, layout and visibility of frame
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}
```

Output :



TEXT AREA

Sometimes a single line of text input is not enough for a given task.

To handle these situations, the AWT includes a simple multiline editor called TextArea

The constructors for TextArea are given below.

- **TextArea( ) :** It creates a new text area with the empty string as text.

TextArea(int numLines, int numChars) : It creates a new text area with the specified number of rows and columns and the empty string as text.

TextArea(String str) : It creates a new text area with the specified text.

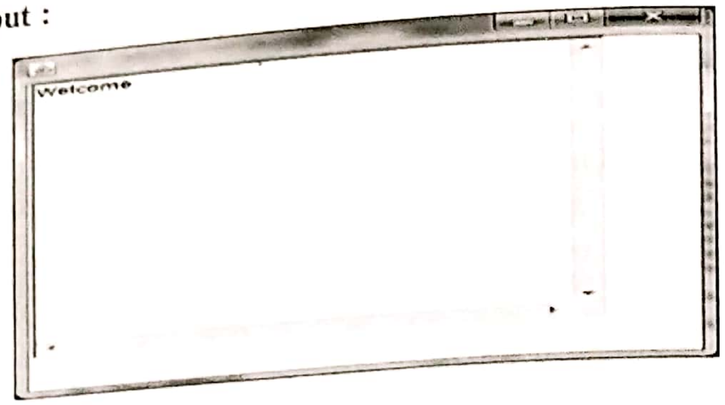
TextArea(String str, int numLines, int numChars) : It creates a new text area with the specified text, and with the specified number of rows and columns.

EXAMPLE

Write a java program to illustrate TextArea.

```
//importing AWT class
import java.awt.*;
public class TextAreaExample
{
// constructor to initialize
  TextAreaExample()
 {
// creating a frame
  Frame f = new Frame();
// creating a text area
  TextArea area = new TextArea("Welcome");
// setting location of text area in frame
  area.setBounds(10, 30, 300, 300);
// adding text area to frame
  f.add(area);
// setting size, layout and visibility of frame
  f.setSize(400, 400);
  f.setLayout(null);
  f.setVisible(true);
 }
public static void main(String args[])
{
  new TextAreaExample();
}
}
```

Output :



MENU AND MENU BAR

A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in Java by the following classes.

- MenuBar.
- Menu.
- MenuItem.

In general, a menu bar contains one (or) more Menu objects.

Each Menu object contains a list of MenuItemobjects.EachMenuItem object represents something that can be selected by the user.

The constructors for Menu are given below.

- **Menu( )**
- **Menu(String optionName)**
- **Menu(String optionName, boolean removable)**

Here, optionName specifies the name of the menu selection.

If removable is true, the pop-up menu can be removed.

Otherwise, it will remain attached to the menu bar.

MenuItem defines these constructors.

- **MenuItem( )**
- **MenuItem(String itemName)**
- **MenuItem(String itemName, MenuShortcutkeyAccel)**

Here, itemName is the name shown in the menu.

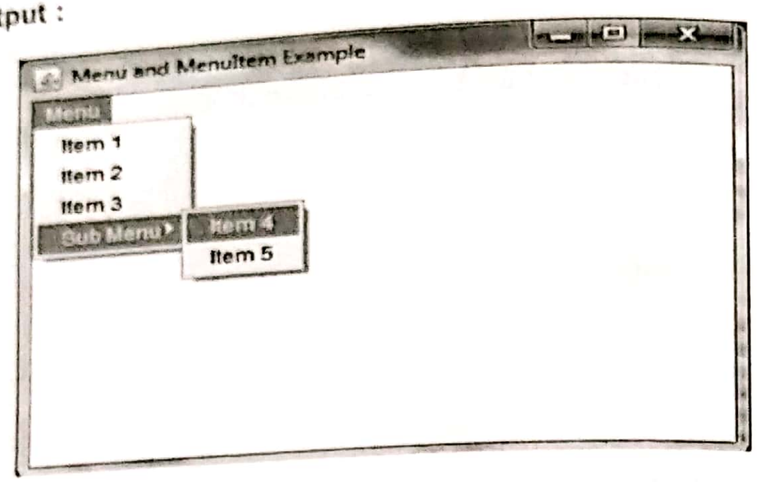
keyAccel is the menu shortcut for this item.

EXAMPLE

Write a java program to illustrate MenuItem and Menu.

```
import java.awt.*;
class MenuExample
{
   MenuExample()
   {
      Frame f= new Frame("Menu and MenuItem Example");
      MenuBar mb=new MenuBar();
      Menu menu=new Menu("Menu");
      Menu submenu=new Menu("Sub Menu");
      MenuItem i1=new MenuItem("Item 1");
      MenuItem i2=new MenuItem("Item 2");
      MenuItem i3=new MenuItem("Item 3");
      MenuItem i4=new MenuItem("Item 4");
      MenuItem i5=new MenuItem("Item 5");
      menu.add(i1);
      menu.add(i2);
      menu.add(i3);
      submenu.add(i4);
      submenu.add(i5);
      menu.add(submenu);
      mb.add(menu);
      f.setMenuBar(mb);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Output :



DIALOG BOX

Dialog boxes are primarily used to obtain user input.

They are similar to frame windows, except that dialog boxes are always child windows of a top-level window.Dialog boxes dont have menu bars.

Two commonly used constructors are shown below.

- **Dialog(Dialog owner) :** It creates an initially invisible, modeless Dialog with the specified owner Dialog and an empty title.
- **Dialog(Dialog owner, String title) :** It creates an initially invisible, modeless Dialog with the specified owner Dialog and title.

Dialog(Dialog owner, String title, boolean modal) : It creates an initially invisible Dialog with the specified owner Dialog, title, and modality.

Commonly used methods.

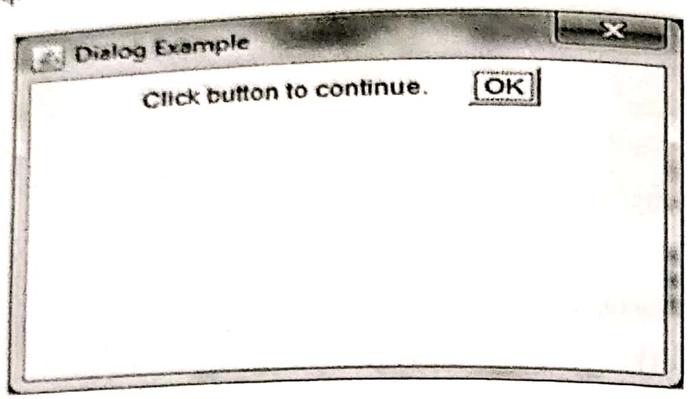
- void **show() :** It shows the dialog.
- String **getTitle() :** It returns the title of the dialog.
- void**setTitle(String title) :** It sets the dialog title.
- boolean **isResizable() :** It returns whether the dialog is resizable.
- **setResizable(boolean flag) :** It sets the dialog to be resizable.

EXAMPLE

Write a java program to illustrate Dialog box.

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample
{
  private static Dialog d;
  DialogExample()
  {
    Frame f= new Frame();
    d = new Dialog(f , "Dialog Example", true);
    d.setLayout( new FlowLayout() );
    Button b = new Button ("OK");
    b.addActionListener ( new ActionListener()
    {
      public void actionPerformed( ActionEvent e )
      {
        DialogExample.d.setVisible(false);
      }
    });
    d.add( new Label ("Click button to continue."));
    d.add(b);
    d.setSize(300,300);
    d.setVisible(true);
  }
  public static void main(String args[])
  {
    new DialogExample();
  }
}
```

Output :



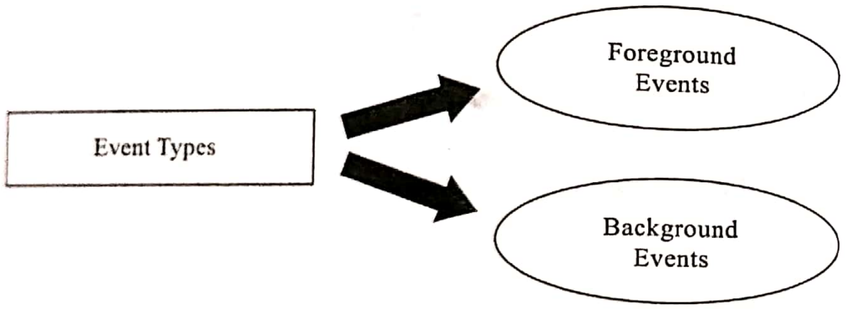
4.5 EVENT HANDLING MECHANISM AND DELEGATION EVENT MODEL

An event can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard (or) page scrolling, etc.,

The **java.awt.event** package can be used to provide various event classes.

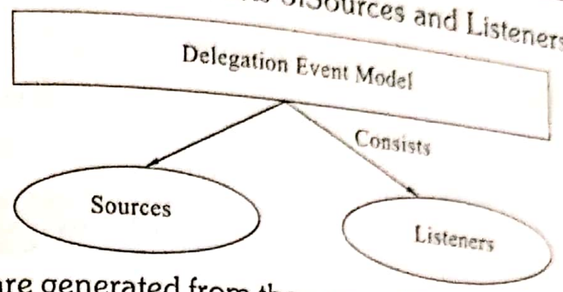
Classification of Events : Events are classified into two types. They are.

- Foreground Events.
- Background Events.



1. **Foreground Events :** Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (**GUI**). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.,
2. **Background Events :** Events that dont require interactions of users to generate are known as background events. Examples of these events are operating system failures/ interrupts, operation completion, etc.,
3. **Event Handling :** It is a mechanism to control the events and to decide what should happen after an event occurs. To handle the events, Java follows the Delegation Event model

4. **Delegation Event Model :** It consists ofSources and Listeners.



5. **Source :** Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.

Source is an object that generates an event.

Whenever user creates any GUI component, component is qualified with value and reference.

For example creation of Button component is given below.

```
Button b=new Button("Submit");
```

Whenever user interacts with any GUI component then JVM creates an object of predefined class for storing value and reference.

The general notation of predefined class is given below.

```
XXXEvent
```

The object of XXXEvent class stores value and reference.

Each GUI component must have one XXXEvent.

S.No.	GUI Component	XXX Event
1.	Button	Action Event

6. **Listeners :** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

Once an event occurs then source has to provide event information to corresponding listener.

Whenever an event is generated then source has to provide event information to corresponding listener.

In AWT, The predefined interfaces are called *listeners*.

The general notation of predefined interface is given below.

```
XXXListener.
```

Each GUI component must have one XXXListener.

S.No.	GUI Component	XXXListener
1.	Button	Action Listener

Each listener consists of abstract/predefined methods.

S.No.	XXXListener	Abstract Method
1.	ActionListener	Public void action performance (Action Event).

To perform Event Handling, we need to register the source with the listener.

Registering the Source with Listener.

Different classes provide different registration methods.

Syntax for registering the Source with Listener :

```
object.addTypeListener();
```

where Type represents the type of event.

For **ActionEvent** we use **addActionListener()** to register.

Flow of Event Handling.

1. User Interaction with a component is required to generate an event.
2. The object of the respective event class is created automatically after event generation, and it holds all information of the event source.
3. The newly created object is passed to the methods of the registered listener.
4. The method executes and returns the result.

The three approaches for performing event handling are by placing the event handling code in one of the below-specified places.

(i) Within Class.

(ii) Anonymous Class.

(iii) Other Class.

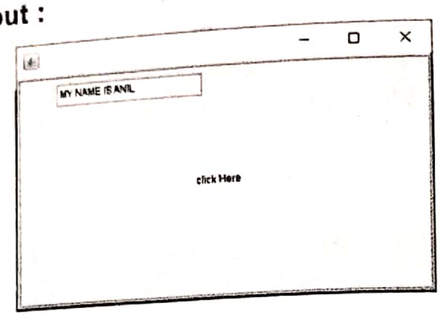
EVENT HANDLING WITHIN CLASS

EXAMPLE-1

Write a java program to demonstrate the event handling within the class.

```
importjava.awt.*;
importjava.awt.event.*;
class EH extends Frame implements ActionListener
{
   TextField t;
   EH()
   {
      // Component Creation
      t = new TextField();
      // setBounds method is used to provideposition and size of the component
      t.setBounds(60, 50, 180, 25);
      Button b = new Button("click Here");
      b.setBounds(100, 120, 80, 30);
       // Registering component with listenerthis refers to current instance
      b.addActionListener(this);
       // add Components
      add(t);
      add(b);
       // set visibility
      setVisible(true);
   }
   // implementing method of actionListener
   public void actionPerformed(ActionEvent e)
   {
      // Setting text to field
      t.setText("MY NAME IS ANIL");
   }
   public static void main(String[] args)
   {
     EH e1=new EH();
   }
}
```

Output :



Explanation :

1. Firstly extend the class with the applet and implement the respective listener.
2. Create Text-Field and Button components.
3. Registered the button component with respective event i.e., ActionEvent by addActionListener().
4. In the end, implement the abstract method.

EVENT HANDLING BY ANONYMOUS CLASS

EXAMPLE-2

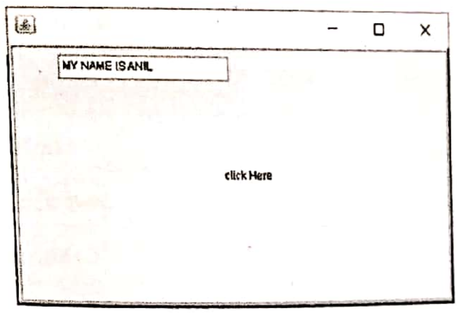
Write a java program to demonstrate event handling by the anonymous class.

```
importjava.awt.*;
importjava.awt.event.*;
class EH3 extends Frame
{
   TextField t;
   EH3()
   {
      // Component Creation
      t= new TextField();
      // setBounds method is used to provide position and size of component
      t.setBounds(60, 50, 180, 25);
      Button b= new Button("click Here");
      b.setBounds(100, 120, 80, 30);
      // Registering component with listener anonymously
      b.addActionListener(new ActionListener()
```



```
      {
      public void actionPerformed(ActionEvent e)
         {
            // Setting text to field
             t.setText("MY NAME IS ANIL");
         }
      });
      // add Components
       add(t);
       add(b);
      // set visibility
       setVisible(true);
   }
   public static void main(String[] args)
   {
      EH3 s=new EH3();
   }
}
```

Output :



EVENT HANDLING BY OTHER CLASS

EXAMPLE-3

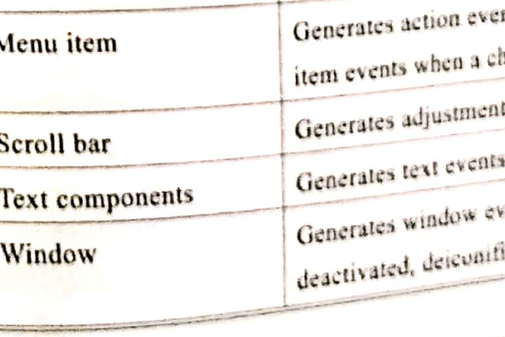
Write a java program to demonstrate the event handling by the other class.

```
importjava.awt.*;
importjava.awt.event.*;
class EHO extends Frame
{
```

```
TextField t;
EHO()
  {
    // Component Creation
    t = new TextField();
    // setBounds method is used to provide position and size of component
t.setBounds(60, 50, 180, 25);
    Button b = new Button("click Here");
b.setBounds(100, 120, 80, 30);
    Other ot = new Other(this);
    // Registering component with listener ,Passing other class as reference
b.addActionListener(ot);
    // add Components
add(t);
add(b);
    // set visibility
setVisible(true);
  }
public static void main(String[] args)
  {
new EHO();
  }
}

// import necessary packages
importjava.awt.event.*;
// implements the listener interface
class Other implements ActionListener
{
  EHO g;
Other(EHO g)
  {
this.g = g;
  }
public void actionPerformed(ActionEvent e)
  {
    // setting text from different class
g.t.setText("MY NAME IS ANIL");
  }
}
```

Output :



4.6 SOURCES OF EVENTS

Some of the user interface components that can generate the events. In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events. For example, you can receive key and mouse events from an applet.

The below table provides different event sources with description.

S.No.	Event Source	Description
1.	Button	Generates action events when the button is pressed.
2.	Check box	Generates item events when the check box is selected or deselected.
3.	Choice	Generates item events when the choice is changed.
4.	List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
5.	Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected (or) deselected.
6.	Scroll bar	Generates adjustment events when the scroll bar is manipulated.
7.	Text components	Generates text events when the user enters a character.
8.	Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, (or) quit.

4.7 EVENT CLASSES AND EVENT LISTENER INTERFACES

The below table provides different event sources and corresponding event listener interfaces with description.

Event Classes in Java

S.No.	Event Class	Event Listener Interface	Description
1.	ActionEvent	ActionListener	An event that indicates that acomponent-defined action occurred like a button click (or) selecting an item from the menu item list.
2.	AdjustmentEvent	AdjustmentListener	The adjustment event is emitted by an Adjustable object like Scrollbar.
3.	ComponentEvent	ComponentListener	An event that indicates that a component moved, the size changed or changed its visibility.
4.	ContainerEvent	ContainerListener	When a component is added to a container (or) removed from it, then this event is generated by a container object.
5.	FocusEvent	FocusListener	These are focus-related events, which include focus, focusin, focusout, and blur.
6.	ItemEvent	ItemListener	An event that indicates whether an item was selected (or) not.
7.	KeyEvent	KeyListener	An event that occurs due to a sequence of keypresses on the keyboard.
8.	MouseEvent	MouseListener&MouseMotionListener	The events that occur due to the user interaction with the mouse (Pointing Device).
9.	MouseWheelEvent	MouseWheelListener	An event that specifies that the mouse wheel was rotated in a component.
10.	TextEvent	TextListener	An event that occurs when an objects text changes.
11.	WindowEvent	WindowListener	An event which indicates whether a window has changed its status (or) not.

Note : As Interfaces contains abstract methods which need to implemented by the registered class to handle events.

Different interfaces consists of different methods which are specified below.

Listener Interfaces

S.No.	Listener Interface	Methods
1.	ActionListener	• actionPerformed()
2.	AdjustmentListener	• adjustmentValueChanged()
	ComponentListener	• componentResized() • componentShown() • componentMoved() • componentHidden()
3.	ContainerListener	• componentAdded() • componentRemoved()
4.	FocusListener	• focusGained() • focusLost()
5.	ItemListener	• itemStateChanged()
6.	KeyListener	• keyTyped() • keyPressed() • keyReleased()
7.	MouseListener	• mousePressed() • mouseClicked() • mouseEntered() • mouseExited() • mouseReleased()
8.	MouseMotionListener	• mouseMoved() • mouseDragged()
9.	MouseWheelListener	• mouseWheelMoved()
10.	TextListener	• textChanged()
	WindowListener	• windowActivated() • windowDeactivated() • windowOpened() • windowClosed() • windowClosing() • windowIconified() • windowDeiconified()

1. **Event Listener Interfaces :** The delegation event model has two parts: sources and listeners. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. The Event Listener Interfaces along with methods are discussed below.

2. **ActionListener Interface :** This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown below.

```
void actionPerformed(ActionEvent ae)
```

3. **AdjustmentListener Interface :** This interface defines the adjustmentValueChanged( ) method that is invoked when an adjustment event occurs. Its general form is shown below.

```
void adjustmentValueChanged(AdjustmentEvent ae).
```

4. **ComponentListener Interface :** This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown below.

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

5. **ContainerListener Interface :** This interface contains two methods. When a component is added to a container, componentAdded( ) is invoked. When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are shown below.

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

6. **FocusListener Interface :** This interface defines two methods. When a component obtains keyboard focus, focusGained( ) is invoked. When a component loses keyboard focus, focusLost( ) is called. Their general forms are shown below.

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

7. **ItemListener Interface :** This interface defines the itemStateChanged( ) method that is invoked when the state of an item changes. Its general form is shown below.

```
void itemStateChanged(ItemEvent ie)
```

8. **KeyListener Interface :** This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively. The keyTyped( ) method is invoked when a character has been entered.

For example, if a user presses and releases the a key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the home key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown below.

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

9. **MouseListener Interface :** This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered() method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown below.

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

10. **MouseMotionListener Interface :** This interface defines two methods. The mouseDragged( ) method is called ***multiple times*** as the mouse is dragged. The mouseMoved( ) method is called ***multiple times*** as the mouse is moved. Their general forms are shown below.

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

11. **MouseWheelListener Interface :** This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown below.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

12. **TextListener Interface :** This interface defines the textValueChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown below.

```
void textValueChanged(TextEvent te)
```

13. **WindowFocusListener Interface :** This interface defines two methods: windowGainedFocus( ) and windowLostFocus(). These are called when a window gains (or) loses input focus.

Their general forms are shown below.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

14. **WindowListener Interface :** This interface defines seven methods. The windowActivated( ) and windowDeactivated() methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified,thewindowDeiconified( ) method is called. When a window is opened (or) closed, the windowOpened( ) or windowClosed() methods are called, respectively. The windowClosing( ) method is called when a window is being closed.

The general forms of these methods are given below.

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

4.8 MOUSE AND KEYBOARD EVENTS

HANDLINGMOUSEEVENTS

To handle mouse events, you have to implement mouse related listeners.

There are 3 listeners related to mouse. They are.

- MouseListener
- MouseMotionListener
- MouseWheelListener

This interface defines five methods.

If the mouse is pressed and released at the same point, mouse Clicked() is invoked.

When the mouse enters a component, the mouseEntered( )method is called.

When mouse leaves from a component, mouseExited( ) is called.

The mousePressed() and mouseReleased( ) methods are invoked when the mouse is pressed and released.

EXAMPLE-1

Write a java program to handle mouse events.

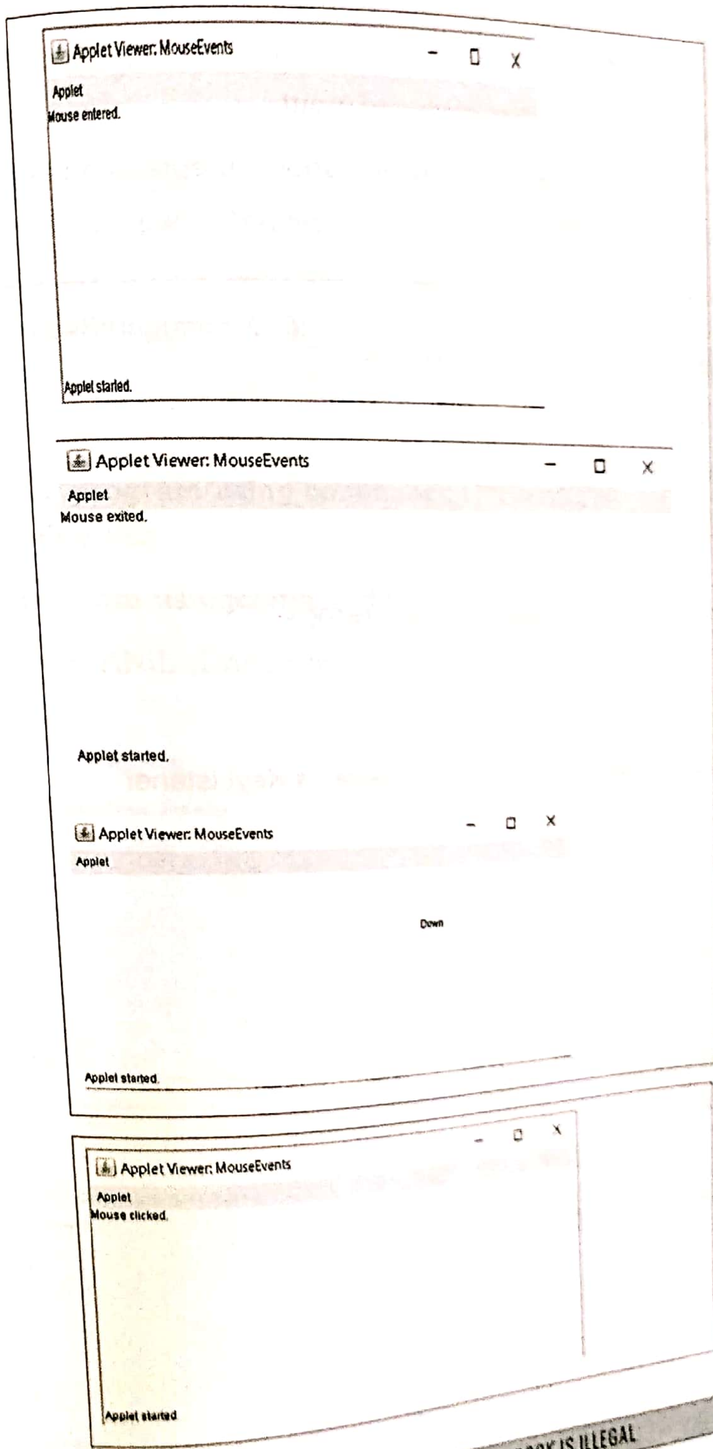
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<appletcode="MouseEvents" width=300 height=100>
</applet>
*/
public class Mouse Events extends Applet implements MouseListener
{
Stringmsg = "";
int mouseX = 0,mouseY = 0;
public void init()
{
addMouseListener(this);
}
public void mouse Entered(MouseEvent me)
{
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
public void mouse Exited(Mouse Event me)
{
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
```

```
repaint();
}
public void mousePressed(Mouse Event me)
{
mouseX=me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
public void mouse Released(MouseEvent me)
{
mouseX=me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
public void mouseClicked(MouseEvent me)
{
mouseX=0;
mouseY=10;
msg = "Mouse clicked.";
repaint();
}
public void paint(Graphicsg)
{
g.drawString(msg,mouseX,mouseY);
}
}
```


Compile program using command : C\Users\CH ANIL\Documents>javac MouseEvents.java

Run program usingcommandC : \Users\CH ANIL\Documents>appletviewer MouseEvents.java

Output :



Handling Keyboard Events : To handle keyboard events, you must implement the KeyListenerinterface.

This interface defines three methods. They are :

- keyPressed( )
- keyReleased()
- keyTyped( ) (when a character has been entered)

The following program demonstrates keyboard input.

It echoes keystrokes to the applet window and shows thepressed/released status of each key in the status window.

EXAMPLE-2

Write a java program to handle keyboard events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener
{
String msg="";
int X = 10, Y = 20;
public void init()
{
addKeyListener(this);
requestFocus();
}
public void keyPressed(KeyEvent ke)
{
showStatus("Key Down");
}
```



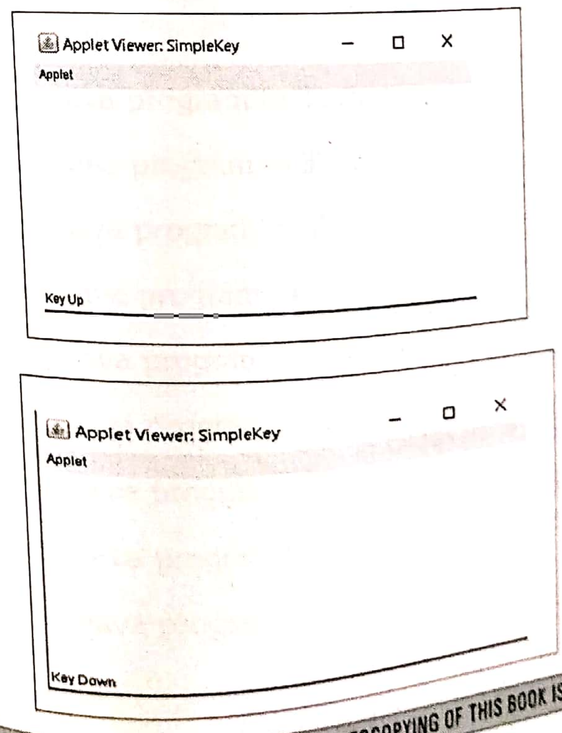
```
public void keyReleased(KeyEvent ke)
{
showStatus("KeyUp");
}
public void keyTyped(KeyEvent ke)
{
msg+=ke.getKeyChar();
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg,X,Y);
}
}
```

Compile program using command : C\Users\CH ANIL\Documents>javac SimpleKey.java

Run program usingcommand C :

\Users\CH ANIL\Documents>appletviewerSimpleKey.java

Output :



REVIEW QUESTIONS

1 Mark Questions

1. Define an applet.
2. List any two advantages of an applet.
3. What is an event ?
4. Define event handling.
5. What is AWT in Java ?
6. What is container in AWT ?
7. What is a source ?
8. What is a listener ?
9. List types of containers provided by AWT.
10. List life cycle methods of an applet.
11. List any two constructors of frame in AWT.
12. List any two constructors of Button.
13. List any two constructors of Checkbox.
14. List any two constructors of List.
15. List any two constructors of Scroll bars.
16. List any two constructors of Text Field.
17. List any two constructors of Text Area.
18. List any two constructors of Dialog box.
19. List any two constructors of Menu.

3 Mark Questions

1. Write the code of an applet skeleton ?
2. What are the five life cycle methods of an applet ?
3. List any three features of AWT in Java.
4. Write the steps involved in event handling.
5. List any three constructors of Label.

6. List any three constructors of Button.
7. List any three constructors of Checkbox.
8. List any three constructors of List.
9. List any three constructors of Scrollbar.
10. List any three constructors of Text Field.
11. List any three constructors of Text Area.
12. List any three constructors of Menu.
13. List any three constructors of Dialog box.
14. List any three AWT Event Classes.

5 Mark Questions

1. Write a java program to demonstrate applet.
2. Explain life cycle of an applet.
3. Explain the usage following AWT controls.
   (a) Label (b) Button (c) Checkbox
   (d) List (e) Scrollbar.
4. Explain the usage of following AWT controls.
   (a) Text Field (b) Text Area (c) Menu bar
   (d) Menu (e) Dialog box.
5. Write a java program to illustrate Label.
6. Write a java program to illustrate Button.
7. Write a java program to illustrate Checkbox.
8. Write a java program to illustrate List.
9. Write a java program to illustrate Scrollbar.
10. Write a java program to illustrate Text Field.
11. Write a java program to illustrate Text Area.
12. Write a java program to illustrate Menu.
13. Write a java program to illustrate Dialog box.

14. Explain Event handling mechanism and Delegation Event model with program.
15. Write short notes on any five Event Listener Interfaces.
16. Write a java program to handle mouse events.
17. Write a Java program to handle keyboard events.