1 a) What is constructor? Mention its types. Explain parameterized constructor with an example [06] CO1 L4

Constructor:

- A **constructor** is a member function of a class which initializes objects of a class.
- It appears as member function of each class whether it is defined or not.
- It is automatically called when an object (instance of class) is created.
- It has the same name as that of the class.
- It may or may not take parameters.
- It does not return any thing, not even void.
- The prototype of a constructor is

  <class name> (<parameter list>);

Types of constructor:

- Different types of constructors are
  1. Zero argument or default constructor
  2. Parameterized constructor
  3. Explicit constructor
  4. Copy constructors

PARAMETERIZED CONSTRUCTORS:

- Parameterized constructors are constructors which takes one or more than one arguments.
- The arguments help to initialize an object when it is created.
- To create a parameterized constructor, we need to add parameters to the constructor.
- Example program to illustrate parameterized constructor

```
// Example program to demonstrate parameterized constructor

#include <iostream>
using namespace std;

class construct
{
        public:
          int a,b;

          // Parameterized constructor
          construct(int x, int y)
          {
                  a = x;
                  b = y;
          }
};
```

```
int main()
{
        // Parameterized constructor called
        construct c(10,20);
        cout << "a = " << c.a << endl;
        cout << "b= " << c.b << endl;

        return 0;
}
```

```
Output:
$ g++ ParamConstructor.C
$ ./a.out
a = 10
b= 20
```

1 b) How do name spaces help in preventing pollution of global namespace [04] CO1 L3

- Namespaces enable the C++ programmers to prevent pollution of the global namespace that leads to name clashes.
- The term 'global namespace' refers to the entire source code. It also includes all the directly and indirectly included header files.
- By default the name of each class is visible in the entire source code, that is, in the global namespace.
- This can lead to problems when a class with the same name is defined in two header files and both the header files are included in a program as shown below

```
// A1.h header file
class A
{
};

// A2.h header file
class A
{
};

// cpp file
#include "A1.h"
#include "A2.h"
void main()
{
        A   Aobj;  // Error: Ambiguity error due to multiple definitions of A
}
```

- To overcome the ambiguity problem, we need to enclose the two definitions of the class in separate **namespaces** and the corresponding namespace followed by the scope resolution operator, must be prefixed to the name of the class while referring to it in the source code as shown below

```
// A1.h header file
namespace A1    // beginning of namespace A1
{
        class A
        {
        };
}               // End of namespace A1


// A2.h header file
namespace A2            // Beginning of namespace A2
{
        class A
        {
        };
}               // End of namespace A2

// cpp file
#include "A1.h"
#include "A2.h"
int main()
{
        A1  :: A   A1obj;       // A1obj is an object of class A defined in A1.h
        A2  :: A   A2obj;       // A2obj is an object of class A defined in A2.h
}
```

2 a) What are static members of a class? Write a C++ program to count the number of objects created [05] CO1 L3

Static member data:

- **A static member of a class is a member that is shared among all objects of the class. It will hold the global data that is common to all objects of the class.**
- The keyword **static** is used to define a member of a class as static.
- When we declare a member of a class as static , no matter how many objects of the class are created, there is only one copy of the static member.
- Static data members are members of the **class** and not members of any **objects** of the class, they are not contained inside any object.
- A statement declaring a static member inside a class will not cause memory to be allocated for it. Therefore, for memory allocation, a static data member must be defined outside the class.
- Static data members can be of any type.
- By default, all static data members of integral type is initialized to zero when the object is created.

Static member function:

- Prefixing the function prototype with the keyword keyword static specifies it as a static member function. The keyword static should not reappear in the definition of the function.

- Static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.
- The static member function's sole purpose is to access and/or modify static data members of the class. They can access only static data members of the class.
- Static member functions do not take the this pointer as a formal argument. Therefore, accessing non-static data members through a static member function results in compile-time error.

```
// Program to count the number of objects created
// using static variable

# include<iostream>
using namespace std;

class A
{
        int code;
        static int count;
        public:
        A ()
        {
                count++;
        }
        void showcount(void)
        {
                cout << "The number of objects created is " << count << endl;
        }

};

int A :: count;

int main()
{
        A obj1, obj2, obj3,obj4;
        obj1.showcount();
}
```

Output:

```
$ g++ static.C
$ ./a.out
The number of objects created is 4
```

2 b) Explain how one can bridge two classes using friend function. Write a C++ program to find the sum of two numbers using bridge friend function add() [05] CO1 L3, L4

- Friend function can be used as bridges between two classes.
- To bridge two classes with a function, the function should be declared s a friend to both the classes.
- Then the friend function can access private data of both class.

-

C++ program to find the sum of two numbers using bridge friend function add()

```
#include <iostream>
using namespace std;

class B ;    // Forward declaration

//void add (A, B);

class A
{
        private:
                int a;

        public:
                A()
                {
                        a = 100;
                }
                friend void add(A,B);
};


class B
{
        private:
                int b;
        public:
                B()
                {
                        b = 200;
                }
                friend void add(A,B);
};

void add (A Aobj, B Bobj)
{
        cout << "Sum of private members of class A and B = " <<  (Aobj.a + Bobj.b);
}

int main()
{
        A A1;
        B B1;
        add (A1, B1);
        return 0;
}
```

Output:

```
$ g++ bridge.C
```

$ ./a.out
Sum of private members of class A and B = 300

3 a) Explain how Delegation Event Model is used to handle events with an example. [10] CO4 L4

THE DELEGATION EVENT MODEL:

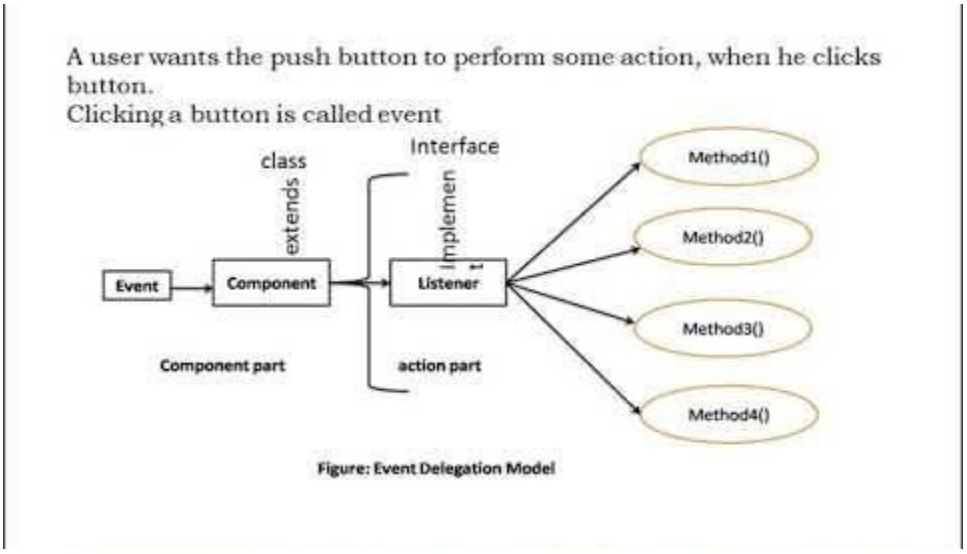


Figure: Event Delegation Model

- The modern approach to handle events is based on the **delegation event model**.
- This model defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a **source** generates an event and sends it to one or more **listeners**.
- The listener simply waits until it receives an event. Once the event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is separated from the user interface logic that generates those events.
- A user interface element is able to "**delegate**" the processing of an event to a separate piece of code.
- In the delegation event model, listeners must **register** with a source in order to receive an event notification.
- This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is more efficient way to handle events than the design used by the old Java 1.0 approach.
- Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

USING THE DELEGATION EVENT MODEL:

- To use the delegation event model follow these two steps:

    - Implement the appropriate interface I the listener so that it will receive the type of event desired.
    - Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- A source may generate several type of events. Each such event is registered separately.
- An object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.
- To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and the keyboard.

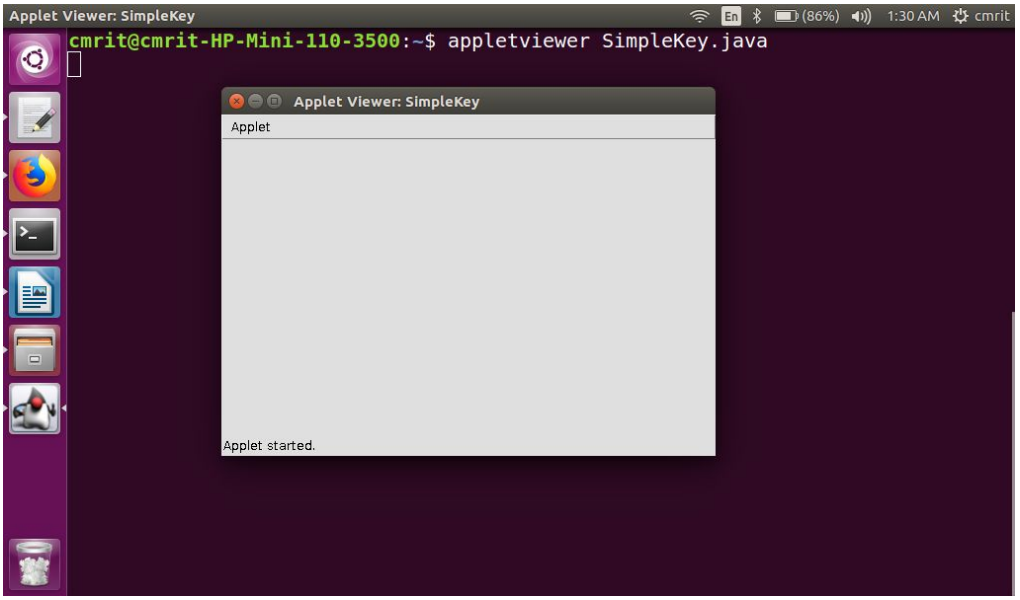
HANDLING KEYBOARD EVENTS:

- When a key is pressed, a KEY_PRESSED event is generated. This results in a call to the keyPressed() event handler.
- When a key is released, a KEY_RELEASED event is generated and the keyReleased() handler is executed.
- If a character is generated by the keystroke, then a KEY_TYPED event is generated and the KeyTyped() handler is invoked.
- Each time a key is pressed, two to three events are generated.
- The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// demonstrate the key event handlers
import java.awt.*;                    // Contains all AWT based event classes used by
Delegation Event Model
import java.awt.event.*;              // Contains all Event Listener Interfaces
import java.applet.*;          // For Applets

/*
<applet code = "SimpleKey" width=500 height=300>
</applet>
*/

public class SimpleKey extends Applet
        implements KeyListener  {
                String msg = "";
                int X = 10, Y = 20;  // Output Coordinates

                // Listener must register with the source. General form -
                // public void addTypeListener (TypeListener el)   Type i s the name of the
event

                public void init()  {
                        addKeyListener(this);
                }

                public void keyPressed(KeyEvent ke)  {
                        showStatus ("key Down");
                }
```

```
        public void keyReleased(KeyEvent ke)  {
                showStatus ("key Up");
        }

        public void keyTyped(KeyEvent ke)  {
                msg += ke.getKeyChar();
                repaint();
        }

        //Display keystrokes
        public void paint(Graphics g)  {
                g.drawString(msg, X, Y);
        }
    }
```

Output:



4 a) Explain Adapter class and inner class with examples [10] CO4 L4

ADAPTER CLASSES:

- Java provides a special feature called an adapter class, which simplifies the creation of event handlers in certain situations.
- **An adapter class provides an empty implementation of all methods in an event listener interface.**
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- We have to define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested.
- For example, the **MouseMotionAdapter** class has two methods, **mouseDragged**() and **mouseMoved**(), which are the methods defined by the **MouseMotionListener** interface. If you are interested only in mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged**(). The empty implementation of **mouseMoved**() would handle the mouse motion events for you.

Table list the commonly used adapter classes in **java.awt.event**

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

Example program to demonstrate an adapter:

- The applet displays a message in the status bar when the mouse is clicked.
- All other mouse events are silently ignored.
- The program has two classes
  1. **AdapterDemo extends Applet.**
     - Its init() method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events.
     - The constructor takes reference to the applet as an argument.
  2. **MyMouseAdapter extends MouseAdapter and overrides the mouseClicked() method.**
     - The other mouse events are silently ignored by code inherited from the MouseAdapter class.

```
// demonstrate an Adapter
import java.awt.*;    // Contains all AWT based event classes used by Delegation Event Model
import java.awt.event.*;    // Contains all Event Listener Interfaces
import java.applet.*;    // For Applets

/*
<applet code = "AdapterDemo" width=500 height=300>
</applet>
*/

public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter (this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter (AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
```

```
        // Handle mouse clicked
        public void mouseClicked(MouseEvent me)  {
                adapterDemo.showStatus("Mouse Clicked");
        }
}
```

INNER CLASSES:

- **Inner class is a a class defined within another class or even within an expression.**
- Inner classes can be used to simplify the code when using event adapter classes.
- Example program to illustrate inner class

Program Explanation:

- The goal of the applet is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.
- InnerClassDemo is a top level class that extends Applet
- MyMouseAdapter is an inner class that extends MouseAdapter.
- As, MyMouseAdapter is defined within the scope of InnerClassDemo, it has access to all the variables and methods within the scope of that class.
- Therefore, the mousePressed() method can call the showStatus() method directly.
- It no longer needs to do this via a stored reference to the applet.
- It is no longer necessary to pass MyMouseAdapter() a reference to the invoking object.

```
// inner class demo
import java.awt.event.*;      // Contains all Event Listener Interfaces
import java.applet.*;         // For Applets

/*
<applet code = "InnerClassDemo" width=500 height=300>
</applet>
*/

public class InnerClassDemo extends Applet {
       public void init()  {
         addMouseListener(new MyMouseAdapter());
       }

       class MyMouseAdapter extends MouseAdapter {   // class within a class
         public void mousePressed(MouseEvent me)  {
                showStatus("Mouse Pressed");
         }
       }
}
```

5 a) What is an Applet? Explain the skeleton of an Applet and five main methods of an Applet.
[10] CO4 L3

APPLET:

- **An applet is a Java code that must be executed within another program. It mostly executes in a Java-enabled web browser.**
- Applets are dynamic and interactive programs. They are usually small in size and facilitate event-driven applications that can be transported over the web**.**

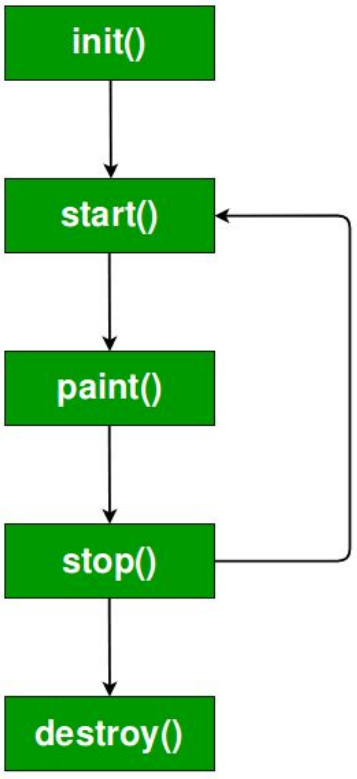
AN APPLET SKELETON:

- In general, applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods, init(), start(), stop() and destroy() apply to all applets and are defined by Applet.
- Default implementations for all of these methods are provided.
- Applets do not need to override those methods they do not use.
- These five methods can be assembled into the skeleton as shown below -

```
// An Applet skeleton
import java.awt.*;
import java.applet.*;

/*
<applet code="AppletSkeleton" width=500 height=300>
</applet>
*/

public class AppletSkeleton extends Applet {
        // Called first.
        public void init()  {
                // initialization
        }

        // Called second, after init().
        // Also called whenever the applet is restarted
        public void start()   {
                // start or resume execution
        }

        // Called when the applet is stopped
        public void stop()  {
                // suspends execution
        }


        // called when applet is terminated.
        // This is the last method executed
        public void destroy  ()  {
                // perform shutdown activities
        }

        // Called when an applets's window must be restored.
        public void paint(Graphics g)  {
                // redisplay contents of window
```

```
    }
}
```

- The following is the order in which the various methods shown in the skeleton are called
- When the applet begins, the following methods are called, in this sequence
  1. init()
  2. start()
  3. paint()
- when an applet is terminated, the following sequence of method calls takes place
  1. stop()
  2. destroy()



init():

- The init() method is the first method to be called.
- We need to initialize variables here.
- This method is called only once during the run time of applet.

start():

- The start method is called after init().
- It is also called to restart an applet after it has been stopped.
- start() is called each time an applet's HTML document is displayed onscreen.
- So, it a user leaves a web page and comes back, the applet resumes execution at start().

paint():

- The paint() method is called each time your applet's output must be redrawn.
- For example, when the applet window may be minimized and then restored.
- Paint() must also be called when the applet begins execution.
- For any reason, whenever the applet must redraw its output, paint() is called.

- The paint() method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.

stop():

- The stop() method is called when a web browser leaves the HTML document containing the applet. For example, when it goes to another page.
- When stop() is called, the applet is probably running.
- We should use stop() to suspend threads that don't need to run when the applet is not visible.
- We can restart them when start() is called if the user returns to the page.

destroy():

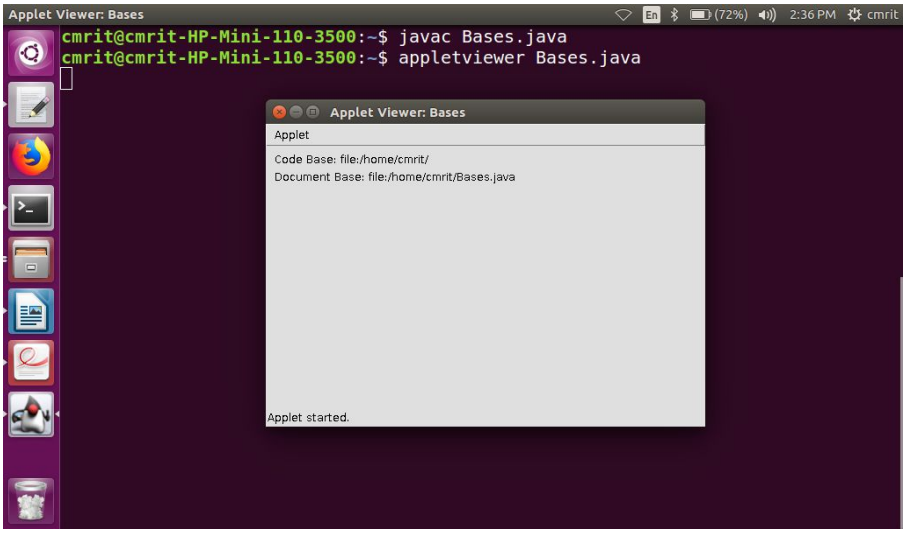
- The destroy() method is called when the environment determines that your applet need to be removed completely from memory.
- At this point, we should free up any resources the applet may be using.
- The stop() method is always called before destroy().

6 a) Explain getDocumentBase() and getCodeBase() in applet class with example program. [5] CO4 L4

- Java allows the applet to load data from the directory holding the HTML file that started the applet (the document base) and the directory from which the applet's class file was loaded (the code base).
- These directories are returned as URL objects by getDocumentBase() and getCodeBase().
- They can be concatenated with a string that names the file you want to load.
- The following applet illustrates these methods.

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;

/*
<applet code="Bases" width=500 height=300>
</applet>
*/

public class Bases extends Applet {
                        // Display code and document bases
        public void paint(Graphics g)  {
                String msg;

                URL url = getCodeBase();
                msg = "Code Base: " + url.toString();
                g.drawString(msg, 10, 20);

                url = getDocumentBase();
                msg = "Document Base: " + url.toString();
                g.drawString(msg, 10, 40);
        }
}
```

Output:



6 b) Demonstrate how to pass parameters for font size and font name in applets.
[5] CO4 L2

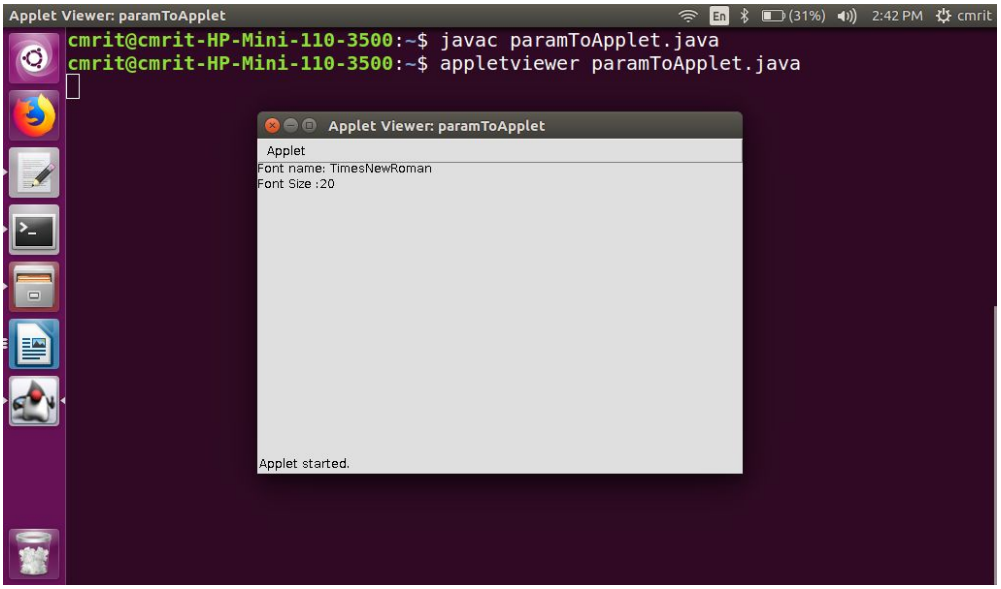
PASSING PARAMETERS TO APPLETS:

- The APPLET tag in HTML allows the user to pass parameters to your applet.
- To retrieve a parameter, use the getParameter() method. It returns the value of the specified parameter in the form of a String object.
- Thus, for numeric and boolean values, we have to convert their string representations into their internal formats.
- Conversions to numeric types must be attempted in a try statement that catches NumberFormatException. Uncaught exceptions should never occur within an applet.
- We should test the return values from getParameter(). If a parameter is not available, getParameter() will return null.
- Example to demonstrate passing parameters

```
// Use Parameters
import java.awt.*;
import java.applet.*;

/*
<applet code="paramToApplet" width=500 height=300>
<param name = fontName value=TimesNewRoman>
<param name = fontSize  value=20>
</applet>
*/

public class paramToApplet extends Applet {
        String fontName;
        int fontSize;

        //Initialize the string to be displayed
```

```
    public void start()  {
        String param;

        fontName = getParameter("fontName");
        if (fontName == null)
            fontName = "Not Found";

        param = getParameter("fontSize");
        try  {
            if (param != null)   // if not found
                fontSize = Integer.parseInt(param);
            else
                fontSize = 0;
        }
        catch (NumberFormatException  e)  {
            fontSize = -1;
        }
    }

    // Display parameters
    public void paint(Graphics g)  {
        g.drawString("Font name: " + fontName, 0, 10);
        g.drawString("Font Size :" + fontSize, 0, 26);
    }
}
```

Output:



7 a) Differentiate between Swings and AWT applets [4] CO5 L3

SWING	AWT

Swing is a set of classes that provides more powerful and flexible GUI components.	AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
Swing supports a pluggable look and feel of the modern Java GUI.	Look and feel of the component is defined by the platform, not by Java.
Swing components are light weight.	AWT components use native code resources hence they are referred to as heavy weight
Swing have main method to execute the program.	AWT need HTML code to run the applet
Swing uses Model-View-Controller (MVC)	Doesn't use MVC

7 b) Explain with syntax
(i) JLabel (ii) JTextField (iii) JButton (iv) JcheckBox [6] CO5 L4

1. Jlabel

- Jlabel is Swing's easiest-to-use component.
- It creates a label. It can be used to display text and/or an icon.
- It is a passive component, it does not respond to user input.
- Jlabel defines several constructors. The three among them are
  - Jlabel(Icon icon)
  - Jlabel(String str)
  - Jlabel(String str, Icon icon, int align)
- str and icon are the text and icon used for the label. The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values
  - LEFT
  - RIGHT
  - CENTER
  - LEADING
  - TRAILING
- These are the constants defined in the SwingConstants interface.

2. JTextField

- **JTextField** is the simplest Swing text component and most widely used text component.
- **JTextField** allows you to edit one line of text.
- It is derived from JTextComponent, which provides the basic functionality common to Swing text components.
- **JTextField** uses the Document interface for its model.
- Three of the **JTextField** constructors are -
  - **JTextField** (int cols)
  - **JTextField** (String str, int cols)
  - **JTextField** (String str)
- str is the string to be initially presented and cols is the number of columns in the text field.
- If no string is specified, the text field is initially empty.
- If the number of columns is not specified, the text field is sized to fit the specified string.

- **JtextField** generates events in response to user interaction. For example, an ActionEvent is fired when the user presses ENTER.
- To obtain the text currently in the text field, call getText().

JButton

- The **JButton** class provides the functionality of a push button.
- Jbutton allows an icon, a string or both to be associated with the push button.
- Three of its constructors are -
  - Jbutton (Icon icon)
  - Jbutton (String str)
  - Jbutton (String str, Icon icon)
- Here, str and icon are the string and icon used for the button.
- When the button is pressed, an ActionEvent is generated.
- Using the ActionEvent object passed to the actionPerformed() method of the registered ActionListener, you can obtain the action command string associated with the button.
- By default this is the string displayed inside the button. We can set the action command by calling setActionCommand() on the button.
- We can obtain the action command by calling getActionCommand() on the event object.

JCheckBox

- The **JCheckBox** class provides the functionality of a check box.
- Its immediate superclass is JtoggleButton which provides support for two-state buttons.
- **JCheckBox** defines several constructors. One among them is
  - JCheckBox(String str)
- It creates a check box that has the text specified by str as a label.
- When the user selects or deselects a check box, an ItemEvent is generated.
- We can obtain a reference t the JCheckBox that generated the event by calling getItem() on the ItemEvent passed t the itemStateChanged() method defiend by ItemListener.
- The easier way to determine the selected state of a check box is to call isSelected() on the JCheckBox instance.