

Applets: Introduction to Applet coding, Applet life cycle, Graphics facility, Color and Font, Passing parameters applets, Appletcontext, Inter Applet Communication.

Threading in Java: Fundamentals of Multi-threading Java coding with Thread classes, Thread Management in Java, Implicit wait, Using Runnable interface, Thread Synchronization, Inter thread communication.

13. Applets in Java	199-220
13.1 Introduction to Applets	199
13.2 Applet Life Cycle	201
13.3 Applet Using Appletviewer	203
13.4 Applet in Browser	204

13.5 Applet with AWT	206
13.6 Important Points for Applet	207
13.7 Font and Color Classes	208
13.8 Graphics Class Facility	211
13.9 Applet Communication	213

14. Threading in Java: Multithreading	221-240
--	----------------

14.1 Multithreading and Multiprocessing	221
14.2 Type of Multitasking	221
14.3 Benefits of Multithreading	222
14.4 Life Cycle of Thread	222
14.5 Main Thread	224
14.6 Creating a Thread	224
14.7 Thread Class Versus Runnable Interface	227
14.8 Thread Methods	228
14.9 Thread Priority	230
14.10 Join() and isAlive() Methods	231
14.11 Synchronization	233
14.12 Interthread Communication	237
14.13 Implicit Wait	239
14.14 Deadlock in java	239



Applets in Java

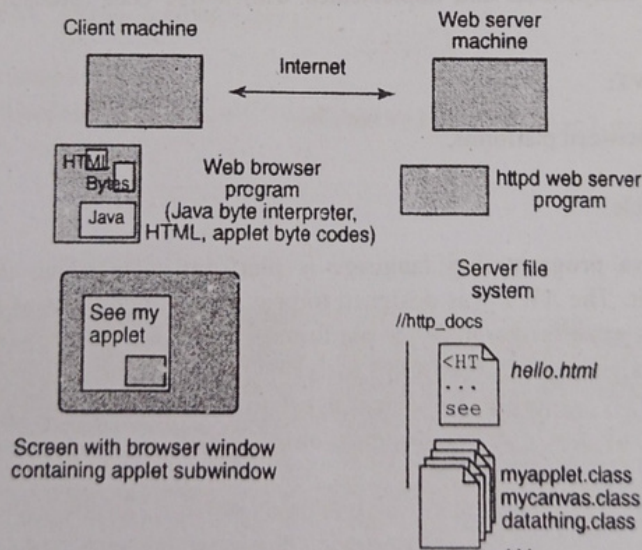
13.1 Introduction to Applets

Applet is a java program which is stored on server machine and its compiled code is travel through internet and execute on client machine. So applets are saved on server machine and execute on client machine with the help of browsers. An *applet* is a Java program that is referenced by a Web page and runs inside a Java-enabled Web browser.

An applet begins execution when an HTML page that “contains” it is loaded. Either a Java-enabled Web browser or the appletviewer is required to run an applet.

The browser detects an applet by processing an `<applet>` HTML tag, which is a paired tag having many attributes.

So applets are small Java programs that are embedded in Web pages. They can be transported over the Internet from one computer (web server) to another (client computers). Even they transform web into rich media and support the delivery of applications via the Internet.



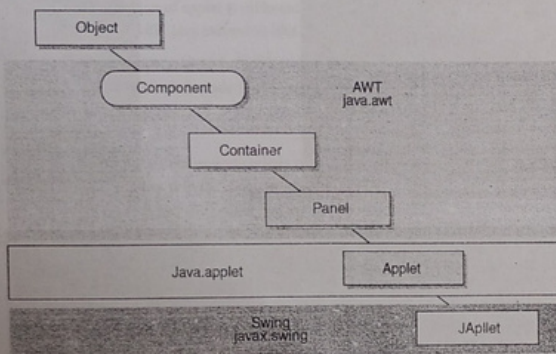
An *applet* is a kind of Java program that can be run across the Internet. Applets are meant to be run from an HTML document.

Applets have many restrictions but they have many features that popularized Java, like:

1. Applets are downloadable executable content.
2. Security policies are easily enforced by browser environment.

3. The most important is that applet provides very beautiful and attractive GUI features than could readily not obtain with just HTML and forms.
4. Applet provides real computations, not just checks on data entry and trivial calculations as in Javascript.

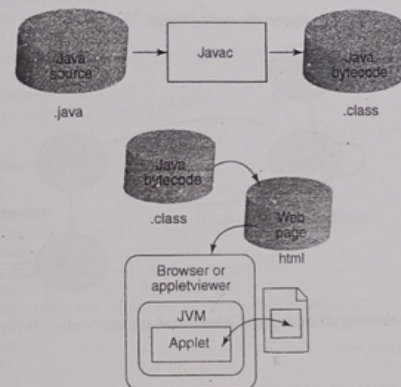
13.1.1 Hierarchy of Applet Class



An applet is runs in a Web browser but applets are fully functional Java application because it has the entire Java API at its disposal. There are certain differences between Applet and Java Standalone Application that are described below:

13.1.2 Difference Between Applet and Java Standalone Application

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

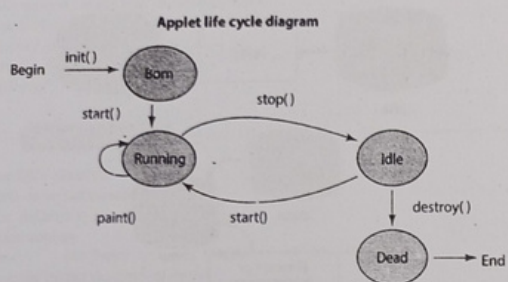


13.1.3 Applet-loading and Starting

1. Browser reads HTML text when encounters "applet" tag.
2. Open connection to server.
3. Fetch byte codes (code for "myapplet.class").
4. Byte codes passed to Java interpreter where get validated by class loader.
5. When page loading complete browser starts Java interpreter.
6. Interpreter opens additional connections to fetch other classes, each loaded by class loader.
7. Applet object starts to run.

13.2 Applet Life Cycle

1. **init() Method:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
2. **start() method:** This method is automatically called after the browser calls the init() method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
3. **stop() method:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
4. **destroy() method:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
5. **paint() method:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.



Example 1: Program showing all methods of life cycle of applet.

```

import java.applet.*;
import java.awt.*;
/*<applet code="MyApp3" width=200 height=200>
</applet>*/
public class MyApp3 extends Applet
{
    public void init()
    {
        System.out.println("init()");
    }
    public void start()
    {
        System.out.println("start()");
    }
    public void paint(Graphics g)
    {
        System.out.println("paint()");
    }
    public void stop()
    {
        System.out.println("stop()");
    }
    public void destroy()
    {
        System.out.println("destroy()");
    }
}
  
```

Output:

```

C:\Users\Savita\Desktop\print_pic> javac MyApp3.java
C:\Users\Savita\Desktop\print_pic> appletviewer MyApp3.java
init()
start()
paint()
stop()
start()
paint()
paint()
paint()
  
```

```

C:\Users\Savita\Desktop\print_pic> appletviewer MyApp3.java
init()
start()
paint()
stop()
start()
paint()
paint()
paint()
destroy()
C:\Users\Savita\Desktop\print_pic>
  
```

13.3 Applet using Appletviewer

Example 2: Applet using Appletviewer window.

```

import java.applet.*;
import java.awt.*;
/*<applet code="MyApplet" width=500 height=200>
</applet>*/
public class MyApp2 extends Applet
{
    public void init()
    {
        setBackground(Color.cyan);
    }
}
  
```

```

setForeground(Color.red);
}
public void paint(Graphics g)
{
g.drawString("Hello",50,25);
}
}

```

Compilation and execution of applet is different from other java program.

First save the program with .java extension like MyApplet.java, and then compile as given below:

```
C:\My_foldername> javac MyApplet.java
```

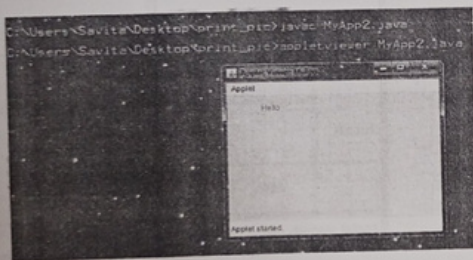
By above javac command whole code is converted in to byte code except comment. Here the code

```
/*<applet code="MyApplet" width=500 height=200>
</applet>*/
```

Contains class file and the commented code is not compiled.
To execute comments, java provides appletviewer command.

```
C:\My_foldername> appletviewer MyApplet.java
```

Output:



13.4 Applet in Browser

Example 3: Applet in browser window.

MyApplet.java

```

import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
public void init()
{
setBackground(Color.cyan);
setForeground(Color.red);
}
}

```

```

public void start(){}
public void paint(Graphics g)
{
g.drawString("Hello",50,25);
}
public void stop(){}
public void destroy(){}
}

```

Index.html

```

<html>
<head>
<title>My First Page</title>
</head>
<body bgcolor=red text=white>
<h1>Welcome to Java Applets</h1>

<applet code="MyApplet" width="200 " height=200></applet><br>
</body>
</html>

```

To create an Applet program follow the steps:

1. Create a Java file containing Applet Code and Methods as described above.
2. Create a HTML file and embed the .class File of the Java file created in the first step
3. Run Applet using either of the following methods
 - o First compile the java file using javac command.
 - o Open the HTML file in java enabled web browser

Output:



13.4.1 Running an Applet Over the Internet

When an applet run in an HTML document and a user at a different location views that HTML document over the Internet, the .class file for the applet is sent over the Internet to enable that user to run the applet on his or her Web browser. This is possible because Java is so portable. A .class file produced on one computer will run on any other computer that has a Java Virtual Machine installed in its Web browser. (Recall that a Java Virtual Machine is the interpreter used to run the byte-code in a Java .class file.)

Note: It is the Java Virtual Machine in the browser that is used to run the applet. Essentially all Web browsers now come equipped with a Java Virtual Machine. So, as long as the browser is installed, the computer on which the browser is run need not have any other Java software such as a Java compiler.

Web browsers do not use the same Java Virtual Machine that is used to run regular Java applications. If there is an old Web browser, it will have an old Java Virtual Machine (or, if it is very old, no Java Virtual Machine). Earlier Java Virtual Machines do not know about JApplets, since an older class used to be used to create applets.

Thus, an old browser, may not be able to run applets from an HTML document. This can be true even if Java applications run fine on your system. The solution for this problem is to obtain a new browser.

Even if there are problems running applets on a browser, and there is no such problem running them from the applet viewer as long as a recent version of Java. So it is possible to run and test applets even if it cannot run from an HTML page.

Typical security settings for Internet Explorer will result in a popup saying some content was disabled click on task bar for more details then can give permission for Applet to load and run.

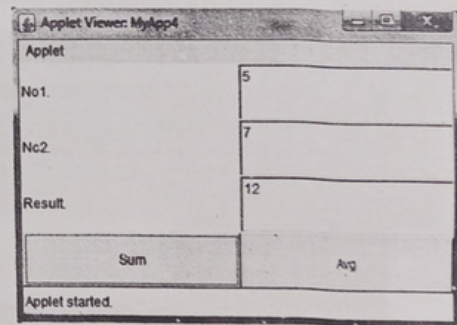
13.5 Applet with AWT

Example 4: Program using Applet with awt.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code="MyApp4" width=400 height=400</applet>*/
public class MyApp4 extends Applet implements ActionListener
{
    Label l1,l2,l3;
    TextField tf1,tf2,tf3;
    Button b1,b2;
    public void init()
    {
        setLayout(new GridLayout(4,2));
        l1=new Label("No1.");
        l2=new Label("No2.");
        l3=new Label("Result.");
        tf1=new TextField();
        tf2=new TextField();
        tf3=new TextField();
        b1=new Button("Sum");
        b2=new Button("Avg");
        b1.addActionListener(this);
        b2.addActionListener(this);
        add(l1).add(tf1);
        add(l2).add(tf2);
    }
}
```

```
add(l3).add(tf3);
add(b1).add(b2);
}
public void actionPerformed(ActionEvent ae)
{
    Button b=(Button)ae.getSource();
    int x=Integer.parseInt(tf1.getText());
    int y=Integer.parseInt(tf2.getText());
    if(b==b1)
        tf3.setText(x+y+"");
    else
        tf3.setText((x+y)/2.0f+"");
}
}
```

Output:



13.6 Important Points for Applet

1. Remove the main method. An applet does not need the things that are typically placed in main.
2. Replace the constructor with a no-parameter method named init() method. The body of the init() method can be the same as the body of the deleted constructor, but with some items removed. The items need to remove are described in the following steps.
 - Delete any invocation of super. The init() method is not a constructor so super cannot be used.
 - Delete any method invocations that serve to program the close-window button of a windowing GUI. An applet has no close-window button.
 - Delete any invocation of setTitle(). Applets have no titles.
 - Delete any invocation of setSize(). Sizing is done by the applet viewer or by the HTML document in which the applet is embedded.

13.7 Font and Color Classes

13.7.1 Font class

To draw text to the screen, it need to create an instance of the *Font* class. Font objects represent an individual font—that is its name, style (bold, italic) and point size. Font names are strings representing the family of the font. Font styles are constants defined by the *Font* class. To create an individual font object use these three arguments to the *Font* class's new constructor:

```
Font f = new Font("TimesRoman", Font.BOLD, 24);
```

This creates a font object for the TimesRoman BOLD font in 24 points. Import the *java.awt.Font* class before use it.

Font styles are actually integer constants that can be added to create combined styles; for example, *Font.BOLD* + *Font.ITALIC* produces a font that is both bold and italic.

```
public void paint(Graphics g)
{
    Font f = new Font("TimesRoman", Font.PLAIN, 72);
    g.setFont(f);
    g.drawString("This is a large font.", 10, 100);
}
```

Example 5: Program with Applets using Font and Graphic class.

```
import java.awt.Font;
import java.awt.Graphics;
/*<applet code="AppFont" width=400 height=400>/>*/

public class AppFont extends java.applet.Applet
{
    public void paint(Graphics g)
    {
        Font f = new Font("TimesRoman", Font.PLAIN, 18);
        Font fb = new Font("TimesRoman", Font.BOLD, 18);
        Font fi = new Font("TimesRoman", Font.ITALIC, 18);
        Font fbi = new Font("TimesRoman", Font.BOLD + Font.ITALIC, 18);

        g.setFont(f);
        g.drawString("This is a plain font", 10, 25);
        g.setFont(fb);
        g.drawString("This is a bold font", 10, 50);
        g.setFont(fi);
        g.drawString("This is an italic font", 10, 75);
        g.setFont(fbi);
        g.drawString("This is a bold italic font", 10, 100);
    }
}
```

Output:

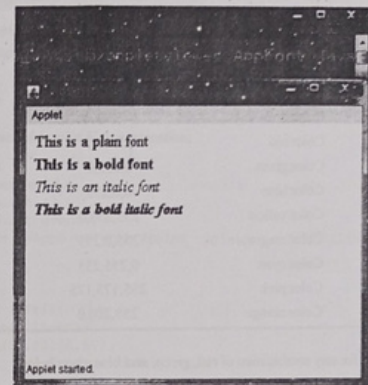


Table: Font methods

Method Name	In Object	Action
getFont()	Graphics	Returns the current font object as previously set by setFont()
getName()	Font	Returns the name of the font as a string
getSize()	Font	Returns the current font size (an integer)
getStyle()	Font	Returns the current style of the font (styles are integer constants: 0 is plain, 1 is bold, 2 is italic, 3 is bold italic)
isPlain()	Font	Returns true or false if the font's style is plain
isBold()	Font	Returns true or false if the font's style is bold
isItalic()	Font	Returns true or false if the font's style is italic

13.7.2 Color Class

Java provides methods and behaviors for dealing with color through *Color* class and also provides methods for setting the current foreground and background colors. Java's abstract color model uses 24-bit color wherein a color is represented as a combination of red, green, and blue values. Each component of the color can have a number between 0 and 255. 0,0,0 is black and 255,255,255 is white and Java can represent millions of colors between as well.

To draw an object in a particular color first create an instance of the *Color* class to represent that color. The *Color* class defines a set of standard color objects stored in class variables to quickly get a color object for some of the more popular colors. Table shows the standard colors defined by variables in the *Color* class.

Table 9.3. Standard colors.

Color Name	RGB Value
Color.white	255,255,255
Color.black	0,0,0
Color.lightGray	192,192,192
Color.gray	128,128,128
Color.darkGray	64,64,64
Color.red	255,0,0
Color.green	0,255,0
Color.blue	0,0,255
Color.yellow	255,255,0
Color.magenta	255,0,255
Color.cyan	0,255,255
Color.pink	255,175,175
Color.orange	255,200,0

It can create a color object for any combination of red, green, and blue given below:

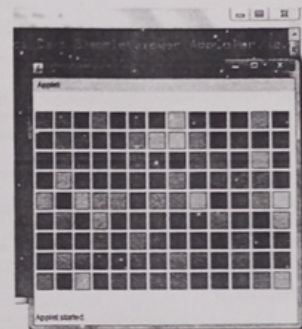
```
Color c = new Color(140,140,240);
```

Example 6: Program with Applets to create a color box using Random class.

```
import java.awt.Graphics;
import java.awt.Color;
/*<applet code="AppColor" width=400 height=400></applet>*/
public class AppColor extends java.applet.Applet
{
    public void paint(Graphics g)
    {
        int rval, gval, bval;

        for (int j = 30; j < (size().height - 25); j += 30)
        for (int i = 5; i < (size().width - 25); i += 30)
        {
            rval = (int)Math.floor(Math.random() * 256);
            gval = (int)Math.floor(Math.random() * 256);
            bval = (int)Math.floor(Math.random() * 256);

            g.setColor(new Color(rval,gval,bval));
            g.fillRect(i, j, 25, 25);
            g.setColor(Color.black);
            g.drawRect(i-1, j-1, 25, 25);
        }
    }
}
```



13.8 Graphics Class Facility

Graphic is a inbuilt abstract class. A Graphics object is usually only obtained as an argument to update and paint method.

```
public void update(Graphics g) {...}
public void paint(Graphics g) {...}
```

The Graphics class provides a set of drawing tools that include methods to draw.

1. **For rectangles:** void fillRect(int x, int y, int w, int h), void drawRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight), void draw3DRect(int x, int y, int w, int h, boolean raised), void draw3DRect(int x, int y, int w, int h, boolean raised), void fill3DRect(int x, int y, int w, int h, boolean raised).
2. **For ovals:** void drawOval(int x, int y, int w, int h), void fillOval(int x, int y, int w, int h).
3. **For arcs:** void drawArc(int x, int y, int w, int h, int startAngle, int arcAngle), void fillArc(int x, int y, int w, int h, int startAngle, int arcAngle).
4. **For polygons:** void drawPolygon(int xPoints[], int yPoints[], int nPoints), fillPolygon(), void drawPolygon(Polygon p), void fillPolygon(int xPoints[], int yPoints[], int nPoints) void fillPolygon(Polygon p).
5. **For rounded rectangles:** drawRoundRect(), void fillRoundRect(int x, int y, int w, int h, int arcWidth, int arcHeight).
6. **For Strings:** drawstring(String).
7. **For images:** drawImage(String).

Example 7: Program with Apples using Graphic class.

```
Index.html:
<html>
<head>
<title>My First Page</title>
</head>
<body bgcolor=red text=white>
<applet code="MyApp8" width=500 height=400>
```



```

<param name="fname" value="arial">
<param name="fsize" value="80">
<param name="red" value="150">
<param name="green" value="50">
<param name="blue" value="100">
</applet><br>
Welcome to Graphic
</body>
</html>

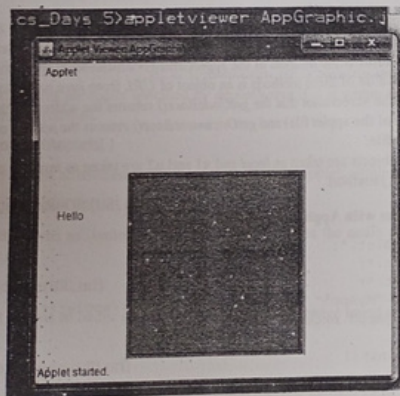
```

```

MyApp8.java
import java.applet.*;
import java.awt.*;
public class MyApp8 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
        g.drawRect(100,100,200,200);
        g.setColor(Color.red);
        g.fillRect(101,101,198,198);
        g.setColor(Color.black);
        g.drawRect(104,104,190,190);
        g.setColor(Color.blue);
        g.drawString("Hello",20,150);
    }
}

```

Output:



13.9 Applet Communication

java.applet.AppletContext class provides the facility of communication between applets. We provide the name of applet through the HTML file. It provides getApplet() method that returns the object of Applet. Syntax:

```
public Applet getApplet(String name){}
```

Example 8: Program of Applet Communication.

```

MyApp6.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class MyApp6 extends Applet implements ActionListener
{
    Button b;

    public void init(){
        b=new Button("Click");
        b.setBounds(50,50,60,50);

        add(b);
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e){

        AppletContext ctx=getAppletContext();
        Applet a=ctx.getApplet("app2");
        a.setBackground(Color.yellow);
    }
}

myapplet.html
<html>
<body>
<applet code="MyApp6.class" width="150" height="150"
name="app1">
</applet>
<applet code="MyApp6.class" width="150" height="150"
name="app2">
</applet>
</body>
</html>

```

The base Java libraries partially support inter-applet communication. However, if there are multiple HTML pages displayed in the browser at one time, the applets in different contexts cannot communicate with one another.

13.9.1 HTML to Applet Communication

<PARAM.> tag is used to pass the parameter value from HTML file to Applet code. In the example shown below parameter which is used is named as "name" and value of the parameter is initialized as "Neeraj". Now in the Applet code you should use same parameter name as "name" to fetch the value.

Sometimes, it may be required to initialize some data while the applet object is being created. This data may be useful to give the properties to the applet object at the creation stage itself or the programmer would like to pass data to the applet. The advantage of this approach is whenever data changes and it require affect in HTML file that does not require compilation. Whatever data is available in the HTML file at the time of applet object creation; it is read and put into the applet. This data is written within the `<param>` tag. To read this data in applet the `getParameter()` method of Applet is use.

The above statement is written within the `<param>` tag. The *name* is used as the key to retrieve the value. The data comes in the form of *name/value* pairs. Name attribute is used to retrieve the value (like *key/value* pairs).

```
String str1 = getParameter("firstNum");
```

`getParameter()` method of Applet takes a string as parameter and returns a string. The parameter is the name of the *name* attribute and the return string is the value associated with the *name* in the `<param>` tag.

The programmer should parse the return string as per requirements. The output is displayed both on the applet window (with `drawString()` method) and the status bar with `showStatus()` method.

Example 9: Program with Applet usint `<PRAM>` tag.

Index.html:

```
<html>
<head>
<title>My First Page</title>
</head>
<body bgcolor=red text=white>
<applet code="MyApp7" width=500 height=400>
<param name="fname" value="arial">
<param name="fsize" value="80">
<param name="red" value="150">
<param name="green" value="50">
<param name="blue" value="100">
</applet><br>
Matrix Computers
</body>
</html>
MyApp6.java
import java.applet.*;
import java.awt.*;
public class MyApp6 extends Applet
{
String fname;
int fntsize,r,g,b;
public void init()
{
fname=getParameter("fname");
if (fname==null)
fname="Courier";
try
{
fntsize=Integer.parseInt(
getParameter("fsize"));
```

```

}
catch (NumberFormatException e)
{
fntsize=10;
}
try
{
r=Integer.parseInt(
getParameter("red"));
g=Integer.parseInt(
getParameter("green"));
b=Integer.parseInt(
getParameter("blue"));
}
catch (NumberFormatException e)
{
r=255;g=0;b=0;
}
Font f1=new Font (fname,Font.PLAIN, fntsize);
setFont (f1);
Color c1=new Color (r,g,b);
setForeground (c1);
}
public void paint (Graphics g)
{
g.setColor (Color.blue);
g.drawString ("Program using <PARAM>", 20, 150);
}
}
```

Applet programmer can get the address of an applet or HTML file using applet methods. The applet method `getCodeBase()` returns the address of the applet and `getDocumentBase()` returns the address of the HTML file. The return value of these methods is an object of `URL` from `java.net` package.

It can notice from the screenshot that the `getCodeBase()` returns the address of the applet file (does not include the name of the of the applet file) and `getDocumentBase()` returns the address of HTML file including the name of the HTML file.

Observe the URL objects are taken as *local* and *u1* and *u2* are taken as *instance* objects because *u1* and *u2* are required in `paint()` method.

Example 10: Program with Applet using URL class.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
/*<applet code="MyApp5" width=400 height=400></applet>*/
public class MyApp5 extends Applet
{
public void init()
{
Font f1=
new Font ("Arial", Font.BOLD, 20);
```

```

setFont (f1);
setBackground (Color.cyan);
setForeground (Color.red);
}
public void paint(Graphics g)
{
URL u1=getCodeBase();
g.drawString ("Code path="+u1,50,25);
URL u2=getDocumentBase();
g.drawString ("Doc. path="+u2,50,75);
}
}

```

13.9.2 AppletCommunication Through AppletContext

AppletContext is the interface defined by *java.applet*. It is used to get information from the applet's execution environment. The context of the currently executing applet is obtained by a call to the `getAppletContext ()` method defined by *Applet*.

Java allows the applet to transfer the control to another URL by using the `showDocument()` Method defined in the *AppletContext* interface. To obtain the Context of the currently executing applet by calling the `getAppletContext ()` method defined by the Applet. Once the context of the applet is obtained with in an applet another document can be brought into view by calling `showDocument ()` method.

There are two `showDocument ()` methods which are as follows:

```

showDocument (URL url)
showDocument (URL url, string loc)

```

where `url` is the object of inbuilt class `URL` from where the document is to be brought into view. `loc` is the location or path within the browser window where the specified document is to be displayed.

13.9.2.1 Methods of AppletContext Class

1. Applet `getApplet(String appName)`

This method returns the applet specified by *appName* if it is within the current applet context. Otherwise *null* is returned.

2. Enumeration `getApplets()`

This method returns an enumeration that contains all of the applets within the current applet context.

3. AudioClip `getAudioClip(URL url)`

This method returns an *AudioClip* object that encapsulates the audio clip found at the location specified by *url*.

4. Image `getImage(URL url)`

This method returns an *Image* object that encapsulates the image found at the location specified by *url*.

5. void `showDocument(URL url)`

This method brings the document at the *URL* specified by *url* into view. This method may not be supported by applet viewers.

6. void `showDocument(URL url, String where)`

This method brings the document at the *URL* specified by *url* into view. This method may not be supported by applet viewers. The placement of the document is specified by *where* as described in the text.

7. void `showStatus(String str)`

This method displays *str* in the status window

Example 11: An applet code to demonstrate the use of `AppletContext` and `showDocument ()`.

```

Index.html
<html>
<head>
<title>My First Page</title>
</head>
<body bgcolor=red text=white>
<applet code="MyApp9"
width=500
height=200</applet><br>
Welcome to AppletContext class
</body>
</html>
MyApp9.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

public class MyApp9 extends Applet implements ActionListener
{
Button b1;
public void init()
{
b1=new Button("Next");
b1.addActionListener(this);
add(b1);
}
public void paint(Graphics g)
{
// g.setColor(Color.red);
g.drawString("Hello",20,150);
}
public void actionPerformed(ActionEvent ae)
{
AppletContext ac=getAppletContext();
try
{
URL u1=getCodeBase();

```

```

URL u2=new URL(u1+"second.html");
ac.showDocument(u2,"_parent");//_blank
}
catch (MalformedURLException e)
{
System.out.println(e.getMessage());
}
}
}

```

Example 12: Program of Applet to Applet communication.

Browser permits the communication between applets. In the following program two applets are created when a button of one applet is clicked the background color of the other applet changes. To create two objects we write two times the <applet> tag within the same <body> tag.

```

Applet1.java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ColorApplet extends Applet implements ActionListener
{
Button pinkBut, orangeBut, cyanBut;
String s1 = "";
AppletContext ctx;
ColorApplet ca;

public void init()
{
pinkBut = new Button("Pink");
orangeBut = new Button("Orange");
cyanBut = new Button("Cyan");

pinkBut.addActionListener(this);
orangeBut.addActionListener(this);
cyanBut.addActionListener(this);

add(pinkBut); add(orangeBut); add(cyanBut);
}
public void start()
{
ctx = getAppletContext();
ca = (ColorApplet) ctx.getApplet(getParameter("abc"));
}
public void actionPerformed(ActionEvent e)
{
String butLabel = e.getActionCommand();

if(butLabel.equals("Pink"))

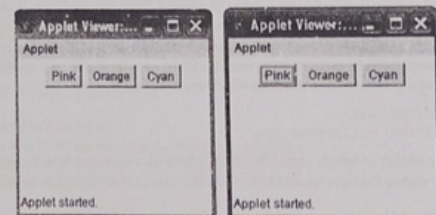
```

```

ca.setBackground(Color.pink);
else if(butLabel.equals("Orange"))
ca.setBackground(Color.orange);
else if(butLabel.equals("Cyan"))
ca.setBackground (Color.cyan);
}
}
index.html
<body>
<applet code="Applet1.class" width="200" height="200" name="first">
<param name="abc" value="second">
</applet>
<applet code = "Applet1.class" width="200" height="200" name="second">
<param name="abc" value="first">
</applet>
</body>

```

Output:

**Very Short Questions**

1. What is applet?
2. Why we uses <applet> tag in html file.
3. List the methods name used in Applet life cycle.
4. What is appletviewer command?
5. Which method is used as main() of applet?
6. What is use of <param> tag?

Short Questions

1. What is applet? Write down uses of applets.
2. Write steps of applet life cycle.
3. Write down note on methods used in applet life cycle.
4. Write difference between applet and java standalone application.

5. Explain execution applet using appletviewer command with java code.
6. Create an applet using <applet> tag in .html file.
7. How java code (applet) communicates with HTML file?

Long Questions

1. What is applet? Write down uses of applets. Explain life cycle of applets.
2. Explain applet communication. Write java code for communication of HTML file to applet.

Threading in Java: Multithreading

14.1 Multithreading and Multiprocessing

A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources. So multithreading is basically used to maximum use of CPU. Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process) which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution. A *process* consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its alone it must be a part of a process.

14.2 Type of Multitasking

There are two distinct types of *Multitasking*.

14.2.1 Processor-Based Multitasking

By definition multitasking is when multiple processes share common processing resources such as a CPU.

14.2.2 Thread-Based Multitasking

Multithreading extends the idea of multitasking where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

As both thread-based and process-based multitasking are types of multitasking but there is very basic difference between the two. *Process-Based Multitasking* is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser. In *Thread-based multitasking*, thread is the small unit of code which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are perform by separate threads.

A multitasking thread requires less overhead than multitasking processor because of the following reasons:

- Processes are heavyweight tasks where threads are lightweight.
- Processes require their own separate address space where threads share the address space.
- Interprocess communication is expensive and limited where Interthread communication is inexpensive and context switching from one thread to the next is lower in cost.

14.3 Benefits of Multithreading

1. Enables programmers to do multiple things at one time.
2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution.
3. Improved performance and concurrency.
4. Simultaneous access to multiple applications.
5. Threads distribute the same address space so reduce the memory use.
6. No process switching takes place which is a much time consuming process.

14.4 Life Cycle of Thread

A thread can be in any of the five following states:

1. **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread. A thread that is just instantiated is in *new* state. When you create a thread with the new operator—for example

```
new Thread(r)
```

The thread is not yet running. This means that it is in the *new* state. When a thread is in the *new* state the program has not started executing code inside of it.

2. **Runnable:** After a newly born thread is started the thread becomes runnable. A thread in this state is considered to be executing its task. Once the start() method is invoke, the thread is *runnable*. A runnable thread may or may not actually be running. It is up to the operating system to give the thread time to run.

Once a thread is running it doesn't necessarily keep running. In fact it is desirable if running threads occasionally pause so that other threads have a chance to run. The details of thread scheduling depend on the services that the operating system provides.

Preemptive scheduling systems give each runnable thread a slice of time to perform its task. When that slice of time is exhausted, the operating system *preempts* the thread and gives another thread an opportunity to work. When selecting the next thread the operating system takes into account the thread *priorities*.

3. **Running State:** It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs

- a. Thread give up its control on its own and it can happen in the following situations
 - i. A thread gets suspended using `suspend()` method which can only be revived with `resume()` method.
 - ii. A thread is made to sleep for a specified period of time using `sleep(time)` method where time in milliseconds.
 - iii. A thread is made to wait for some event to occur using `wait()` method. In this case a thread can be scheduled to run again using `notify()` method.
- b. A thread is pre-empted by a higher priority thread.

A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs. A thread moves out of the blocked state and back into the runnable state by one of the following pathways.

- If a thread has been put to sleep the specified number of milliseconds must expire.
- If a thread is waiting for the completion of an input or output operation then the operation must have finished.
- If a thread is waiting for a lock that was owned by another thread then the other thread must relinquish ownership of the lock.

- If a thread waits for a condition then another thread must signal that the condition may have changed.
- If a thread has been suspended then someone must call its `resume()` method. As the `suspend()` method has been deprecated the `resume` method is also deprecated as well.

A blocked thread can only re-enter the runnable state through the same route that blocked it in the first place.

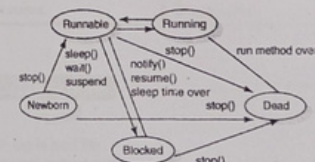
4. **Waiting/Blocking stage:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

A thread enters the *blocked* state when one of the following actions occurs:

- The thread goes to sleep by calling the `sleep()` method.
- The thread calls an operation that is *blocking on input/output*, that is an operation that will not return to its caller until input and output operations are complete.
- The thread tries to acquire a lock that is currently held by another thread.
- The thread waits for a condition.
- Someone calls the `suspend()` method of the thread. However, this method is deprecated, and it should avoid calling deprecated methods. When a thread is blocked (or when it dies) another thread can be scheduled to run. When a blocked thread is reactivated the scheduler checks to see if it has a higher priority than the currently running thread. If so it preempts the current thread and picks a new thread to run.

5. **Terminated/Dead:** A runnable thread enters the terminated state when it completes its task or otherwise terminates. A thread is dead for one of two reasons:

- It dies a natural death because the `run()` method exits normally.
- It dies abruptly because an uncaught exception terminates the `run()` method. A thread can kill by invoking its `stop()` method. That method throws a `Thread Death` error object that kills the thread. The `stop()` method is also deprecated.



The life cycle of threads in Java is very similar to the life cycle of processes running in an operating system. During its life cycle the thread moves from one state to another depending on the operation performed by it.

At any given time a thread can be in only one state. These states are JVM states as they are not linked to operating system thread states.

When the object of a user `Thread` class is created the thread moves to the *NEW* state. After invocation of the `start()` method the thread shifts from the *NEW* to the ready (*RUNNABLE*) state and is then dispatched to running state by the JVM thread scheduler. After gaining a chance to execute the `run()` method will be invoked. It should be noted that when the `run()` method is invoked directly (explicitly in the program), the code in it is executed by the current thread and not by the new thread. Depending on program operations or invocation of methods such as `wait()`, `sleep()`, and `suspend()` or an I/O operation, the thread moves to the *WAITING*, *SLEEPING*, and *BLOCKED* states respectively. It should be noted that the thread is still alive. After the completion of an

operation that blocked it or receiving an external signal that wakes it up, the thread moves to ready state. Then the JVM thread scheduler moves it to running state in order to continue the execution of remaining operations. When the thread completes its execution it will be moved to *TERMINATED* state. A dead thread can never enter any other state not even if the `start()` method is invoked on it.

14.5 Main Thread

Every time a Java program starts up, one thread begins running which is called as the *main thread* of the program because it is the one that is executed when program begins.

- Child threads are produced from main thread.
- Often it is the last thread to finish execution as it performs various shut down operations.

14.6 Creating a Thread

Threads are objects in the Java language. They can be created by using two different mechanisms

- Create a class that extends the standard *Thread* class (By extending the *Thread* class.)
- Create a class that implements the standard *Runnable* interface (By implementing the *Runnable* interface.)

14.6.1 Create Thread by Extending Thread

The first way to create a thread is to create a new class that extends *Thread* and then to create an instance of that class. The *extending class must override the `run()` method* which is the entry point for the new thread. It must also call `start()` to begin execution of the new thread.

The steps for creating a thread by using the first mechanism are:

1. Create a class by extending the *Thread* class and override the `run()` method:

```
class MyThread extends Thread {
public void run() {
// thread body of execution
}
}
```

2. Create a thread object:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of Created Thread:

```
thr1.start();
```

An example program illustrating creation and invocation of a thread object is given below:

Example 1: Program to create thread by extending *Thread* class.

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<=5;i++)
System.out.println("child thread"+i);
}
```

```
System.out.println("end of Thread");
}
}
class MyThread_1
{
public static void main(String arg[])
{
MyThread t1=new MyThread();
t1.start();
for(int i=0;i<=5;i++)
System.out.println("main thread"+i);
System.out.println("end ofmain");
}
}
```

Output:

```
main thread0
main thread1
main thread2
child thread0
child thread1
child thread2
child thread3
child thread4
child thread5
end of main
child thread0
child thread1
child thread2
child thread3
child thread4
child thread5
end of Thread
Press any key to continue...
```

The class *A test* extends the standard *Thread* class to gain thread properties through inheritance. The user needs to implement their logic associated with the thread in the `run()` method which is the body of thread. The objects created by instantiating the class *A Test* are called threaded objects. Even though the execution method of thread is called `run()`, we do not need to explicitly invoke this method directly.

When the `start()` method of a threaded object is invoked, it sets the concurrent execution of the object from that point onward along with the execution of its parent thread/method.

14.6.2 Create Thread by Implementing *Runnable*

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

```
public void run()
```

Here define the code that constitutes the new thread inside `run()` method. It is important to understand that `run()` can call other methods use other classes and declare variables just like the main thread can.

After creating a class that implements *Runnable*, instantiate an object of type *Thread* from within that class. *Thread* defines several constructors. The one that will use is shown here:

Thread(Runnable threadOb, String threadName);

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName. After the new thread is created it will not start running until call its *start()* method which is declared within Thread. The start() method is shown here:

void start();

The steps for creating a thread by using the second mechanism are:

1. Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
    ...
    public void run()
    {
        // thread body of execution
    }
}
```

2. Creating Object:

```
MyThread m1 = new MyThread();
```

Here m1 have run() but not start() method so m1 is not a complete thread in itself.

3. Creating Thread Object:

```
Thread t1 = new Thread(m1);
```

Here t1 is complete thread as it have run() and start() methods.

4. Start Execution:

```
t1.start();
```

Example 2: Program to create thread by implementing Runnable Interface.

```
class MyThread implements Runnable
{
    public void run()
    {
        for(int i=0;i<=5;i++)
            System.out.println("child thread"+i);
    }
}
class MyThread_2
{
    public static void main(String arg[])
    {
        MyThread m1=new MyThread();
        Thread t1=new Thread(m1);
        t1.start();
    }
}
```

```
for(int i=0;i<=5;i++)
System.out.println("main thread"+i);
System.out.println("end ofmain");
}
```

Output:

```
main thread
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
Press any key to continue...
```

The class MyThread implements standard Runnable interface and overrides the run() method and that includes logic associated with the body of the thread. The objects created by instantiating the class MyThread are normal objects. To create it need to create a generic Thread object and pass MyThread object as a parameter to this generic object. As a result of this association, threaded object is created. In order to execute this threaded object, we need to invoke its start() method which sets execution of the new thread.

14.7 Thread Class Versus Runnable Interface

It is a little confusing why there are two ways of doing the same thing in the threading API. It is important to understand the implication of using these two different approaches. By extending the Thread class, the derived class itself is a thread object and it gains full control over the thread life cycle.

Implementing the Runnable interface does not give developers any control over the thread itself as it simply defines the unit of work that will be executed in a thread.

Another important point is that when extending the Thread class, the derived class cannot extend any other base classes because Java only allows single inheritance.

By implementing the Runnable interface the class can still extend other base classes if necessary. To summarize if the program needs a full control over the thread life cycle extending the Thread class is a good choice and if the program needs more flexibility of extending other base classes implementing the Runnable interface would be preferable.

14.8 Thread Methods

S.N.	Methods with Description
1.	public void start() Starts the thread in a separate path of execution then invokes the run() method on this Thread object.
2.	public void run() If this Thread object was instantiated using a separate Runnable target the run() method is invoked on that Runnable object.
3.	public final void setName(String name) Changes the name of the Thread object. There is a getName() method for retrieving the name.
4.	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5.	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6.	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7.	public void interrupt() Interrupts this thread causing it to continue execution if it was blocked for any reason.
8.	public final boolean isAlive() Returns true if the thread is alive which is any time after the thread has been started but before it runs to completion.

Note: A thread cannot start twice if a thread is started it can never be started again if you do so there is an exception that is `IllegalThreadStateException` is thrown.

Example 3: Program creating multiple threads.

```
class ATest extends Thread
{
    ATest(String nm)
    {
        setName(nm);
        start();
    }

    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println(getName()+i);
        }
    }
}
```

```
    }
    System.out.println("End of "+getName());
}

}
class Thread_4
{
    public static void main(String arg[])
    {
        ATest a1=new ATest("First ");
        ATest a2=new ATest("second ");
        ATest a3=new ATest("third ");

        for(int i=1;i<=5;i++)
        {
            System.out.println("Main "+i);
        }
        System.out.println("End of main");
    }
}
```

Output:

```

Main 1
First 1
second 1
second 2
First 2
Main 2
First 3
second 3
third 1
second 4
First 4
Main 3
First 5
second 5
third 2
End of second
End of First
Main 4
third 3
Main 5
third 4
End of main
third 5
End of third
Press any key to continue...
```

14.9 Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. It is important for different threads to have different priorities. Important threads should always have higher priority than less important ones. For example the garbage collector thread just needs the lowest priority to execute which means it will not be executed before all other threads are scheduled to run. It is possible to control the priority of the threads by using the Java APIs.

The `Thread.setPriority(...)` method serves this purpose. The `Thread` class provides 3 constants value for the priority:

```
MIN_PRIORITY = 1,
NORM_PRIORITY = 5,
MAX_PRIORITY = 10
```

Priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default every thread is given priority `NORM_PRIORITY` (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Example 4: Program showing to alter the order of the thread by changing its priority.

```
class Priority_1
{
    public static void main(String arg[])
    {
        Thread t1=Thread.currentThread();
        System.out.println("Current Thread "+t1);
        System.out.println(" "+t1.getName());
        System.out.println(" "+t1.getPriority());
        t1.setName("New_Thread");
        t1.setPriority(Thread.MAX_PRIORITY);
        System.out.println("After setting new data to thread "+t1);
        System.out.println(" "+t1.getName());
        System.out.println(" "+t1.getPriority());
        System.out.println();
        for(int i=0;i<=5;i++)
        {
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException
            e){}

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

Output:

```
Current Thread Thread[5,main]
New_Thread
MAX_PRIORITY
After setting new data to thread Thread[10,main]
New_Thread
MAX_PRIORITY

0
1
2
3
4
5
```

The `sleep()` method causes the current thread to sleep for a specified amount of time in milliseconds:
public static void sleep(long millis) throws InterruptedException
 In above program, the code puts the thread in sleep state for 1second:

```
try
{
    Thread.sleep(1000); // thread sleeps for 1 seconds
} catch (InterruptedException ex){}
```

14.10 Join() and isAlive() Methods

The `isAlive()` method returns true if the thread upon which it is called has been started but not moved to the dead state:

```
public final boolean isAlive()
```

When a thread calls `join()` on another thread the currently running thread will wait until the thread it joins with has completed. It is also possible to wait for a limited amount of time instead for the thread completion.

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

Example 5: Program using `join()` and `isAlive()` method.

```
class ATest implements Runnable
{
    Thread t;
```

```

ATest(String nm)
{
    t=new Thread(this);
    t.setName(nm);
    t.start();
}

public void run()
{
    for(int i=1;i<=5;i++)
    {
        System.out.println(t.getName()+i);
    }
    System.out.println("End of "+t.getName());
}

}

class TJoin_6
{
    public static void main(String arg[])
    {
        ATest a1=new ATest("First ");
        ATest a2=new ATest("second ");
        ATest a3=new ATest("third ");
        System.out.println(a1.t.isAlive());
        System.out.println(a2.t.isAlive());
        System.out.println(a3.t.isAlive());
        try
        {
            a1.t.join();
            a2.t.join();
            //a3.t.join();
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
        System.out.println(a1.t.isAlive());
        System.out.println(a2.t.isAlive());
        System.out.println(a3.t.isAlive());
        for(int i=1;i<=5;i++)

            System.out.println("Main "+i);

        System.out.println("End of main");
    }
}

```

Output:

```

true
true
true
first 1
first 2
first 3
first 4
first 5
first 6
End of First
third 1
third 2
third 3
third 4
third 5
End of third
second 1
second 2
second 3
second 4
second 5
End of second
false
false
false
main 1
main 2
main 3
main 4
main 5
End of main
Press any key to continue...

```

14.11 Synchronization

When two or more threads need access to a shared resource they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization. Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource. Thought synchronization decrease the speed of software but some it is necessary to achieve the perfect accuracy.

14.11.1 Use of Synchronization

In most practical multithreaded applications two or more threads need to share access to the same objects. What happens if two threads have access to the same object and each calls a method that modifies the state of the object? To avoid corruption of shared data by multiple threads synchronization is must.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

14.11.2 Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

synchronized(object)

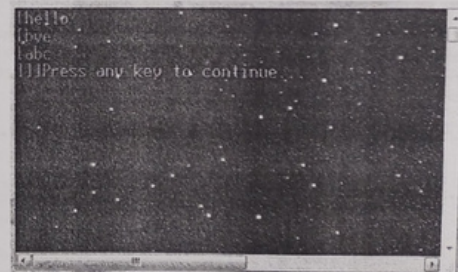
```
{
// statements to be synchronized
}
```

Example 6: Program showing problem without using Synchronization.

```
class A
{
public void display(String msg)
{
System.out.println(""+msg);
try
{
Thread.sleep(1000);
}
catch(Exception e){}
System.out.print("");
}
}
class ATest implements Runnable
{
Thread t;
A obj;
String msg;
ATest(String msg, A obj)
```

```
{
t=new Thread(this);
this.obj=obj;
this.msg=msg;
t.start();
}
public void run()
{
obj.display(msg);
}
}
class ThreadTest1
{
public static void main(String arg[])
{
A a1= new A();
ATest m1=new ATest("hello",a1);
ATest m2=new ATest("bye",a1);
ATest m3=new ATest("abc",a1);
}
}
```

Output:



Due to threading a single method is shared by all threads so before completion of one thread coding another thread call this method so output is not in sequence.

Example 7: Program using synchronization.

```
class A
{
synchronized public void display(String msg)
{
System.out.print(""+msg);
}
```

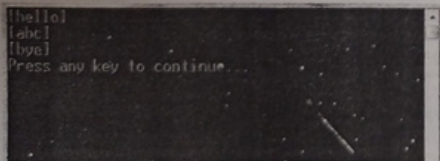
```

try
{
    Thread.sleep(1000);
}
catch(Exception e){

    System.out.println("");
}
}
class ATest implements Runnable
{
    Thread t;
    A obj;
    String msg;
    ATest(String msg, A obj)
    {
        t=new Thread(this);
        this.obj=obj;
        this.msg=msg;
        t.start();
    }
    public void run()
    {
        obj.display(msg);
    }
}
class ThreadTest1
{
    public static void main(String arg[])
    {
        A a1= new A();
        ATest m1=new ATest("hello",a1);
        ATest m2=new ATest("bye",a1);
        ATest m3=new ATest("abc",a1);
    }
}

```

Output:



```

[hello]
[abc]
[bye]
Press any key to continue...

```

Due to synchronization output is in sequence. As the display() method is synchronized no another thread can call it before its completion by one thread.

14.12 Interthread Communication

It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

- **wait()**: This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify()**: This method wakes up the first thread that called wait() on the same object.
- **notifyAll()**: This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

These methods are implemented as final methods in Object so all classes have them. All three methods can be called only from within a synchronized context.

Example 8: Program showing interthread communication.

```

class MyThread extends Thread
{
    private boolean flag;
    MyThread(String nm)
    {
        setName(nm);
    }
    public void run()
    {
        for(int i=0;i<=5;i++)
            System.out.println("child"+getName()+i);
        try
        {
            Thread.sleep(1000);
            synchronized(this)
            {
                while(flag)
                {
                    wait();
                }
            }
        }
        catch(InterruptedException e){}
    }
    synchronized void mySuspend()
    {
        flag=true;
    }
    synchronized void myResume()
    {
        flag=false;
        notify();
    }
}

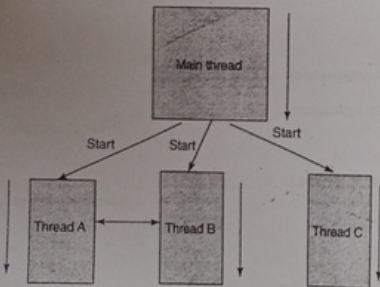
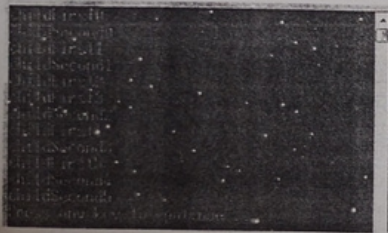
```

```

class ThreadTest5
{
    public static void main(String args[]) throws InterruptedException
    {
        MyThread t1 = new MyThread("First");
        MyThread t2 = new MyThread("Second");
        t1.start();
        t2.start();
        Thread.sleep(1000);
        t1.mySuspend();
        Thread.sleep(1000);
        t1.myResume();

        Thread.sleep(1000);
        t2.mySuspend();
        Thread.sleep(1000);
        t2.myResume();
    }
}
    
```

Output:



14.13 Implicit Wait

Some elements take some time to appear on software when it is loading. If implicit wait is applied in test case then web driver will wait for specified amount of time if targeted element not appears on page then it wait for a certain amount of time before throwing an *exception*

Implicit wait means that it wait for a certain amount of time before throwing an *exception* that it cannot find the element on the page. We should note that implicit waits will be in place for the entire time the browser is open. This means that any search for elements on the page could take the time the implicit wait is set for.

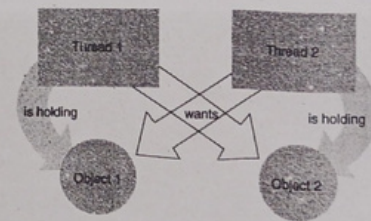
Syntax- driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

Here it will wait for 10sec if while execution driver did not find the element in the page immediately. This code will attach with each and every line of the script automatically. It is not required to write every time. Just write it once after opening the browser. If element found before assigned value it will continue with the script it will not wait for completely 10 sec.

14.14 Deadlock in java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them and then release the lock when it's done with them.

In java a deadlock is a situation where minimum two threads are holding lock on some different resource and both are waiting for other's resource to complete its task. And none is able to leave the lock on resource it is holding.



Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since both threads are waiting for each other to release the lock the condition is called deadlock.

```

ClassA1 a1 = new ClassA1();
ClassA2 a2 = new ClassA2();
synchronized void m1(a2)
{
    ...
    a2.m2(a1);
    ...
}
synchronized void m2(a1)
{
    ...
    a1.m1(a1);
    ...
}
    
```

Very Short Questions

1. What is multiprocessing?
2. What is multithreading?
3. Which method is main() of thread?
4. Which method inserts thread in CPU queue?
5. How many methods are declared in Runnable interface?
6. Describe Runnable interface verses Thread class.
7. Name the thread methods.

Short Questions

1. What is multithreading? Write benefits of multithreading.
2. Write short note on thread life cycle.
3. Write syntax to create thread using Thread class.
4. Write syntax to create thread using Runnable interface.
5. What is synchronization?
6. What do you mean by inter thread communication? Explain.
7. Explain deadlock using java code.

Long Questions

1. What is multithreading? Write benefits of multithreading. Explain thread life cycle in detail.
2. Explain the various methods to create thread using java code.
3. Explain thread management in java with synchronization.

