THARANESWARAN VELMURUGAN
2019239025
M.S.C Computer Science
Java Theory Assignment.

Applet:

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.

Advantages:

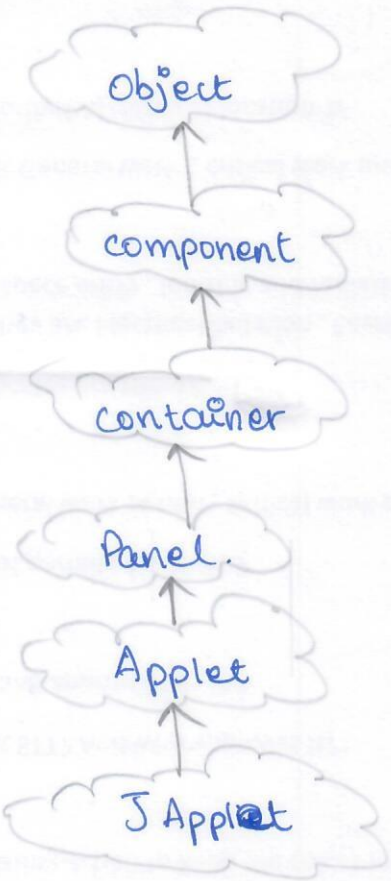
* There are many advantages of applet. They are as follows:

(i) It works at client side so less response time

(ii) It is secured

(iii) It can be executed by browers running under many platforms including linux, windows, Mac Os etc

Hierachy of Applet



java.applet.Applet class :

(i) public void init() :

It used to initialized the Applet.
It is invoked only once

(ii) public void start() :

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

(iii) public void stop() :

It is used to stop the Applet . It is invoked when Applet is stop or browser is minized.

(iv) Public void destroy() :

Is used to destroy the Applet . It is invoked only once.

java . awt . Component class :

The component class provides 7 life cycle methods of applet.

(i) public void paint (Graphics g) :

* It is used to paint the Applet . It provides graphics class object that can be used for drawing oval , rectangle , arc etc .

Threads:

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program

Java Thread Benefits:

* Java threads are light weight compared to processed, it takes less time and resource to create a thread.
* Threads share their parent process data and code
* context switching between threads is usually less expensive than between processes
* Thread intercommunication is relatively easy than process communication
* It increases the responsiveness of GUI applications
* Better utilization of system resources.
* Simplify program logic when there are multiple independent entities.

Creating the thread:

Two ways:

* Extending the thread class
* Implementing the Runnable interface

The thread class:

* Extend the thread class
* Override the run() method.
* Create an object of the sub class and call the start method to excute the thread

The Runnable Interface.

* useful when the class is already extending another class and needs to implement multithreading
* Need to implement the run() method.
* create a threads object and give it a Runnable object.

→ Public Thread (Runnable target);

* Start the thread by calling the start() method.

Thread States:

(i) Newborn State:

When the new instance of the thread is created by executing the constructor thread(), Thread is said to be newborn. In this state no resources are allocated to the thread.

(ii) Runnable State:

When a start() method is called on the newborn thread instance thread makes the transition to Runnable State. In this state the thread is waiting for the schedular to Schedule it on the processor.

(iii) Running State:

When thread is being executed by the CPU it will be in the running state.

(iv) Non-Runnable State:

A running state can be suspended, ie temporarily suspended from its activity. This is done by invoking sleep() or wait() method.

A suspended thread can then be resumed allowing it to pick up from where it left off, by calling notify() method or when the sleep interval expires.

Dead State:

A thread comes to a dead state if it is over with the task.

Thread Scheduling in JAVA:

* Thread scheduler in Java is the part of the JVM that decides which thread should run.
* There no guarantee that which runnable thread will be chosen to run by the thread Scheduler.
* Only one thread at a time can run in a single process.
* The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.
* under preemptive scheduling the highest priority task should execute next, based on priority and other factors.

* Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence

* Under time slicing, a task executes defined slice of time and then re-enters the pool of ready tasks.

* The scheduler then determine which task should execute next, based on priority and other factors.

* JVM's thread scheduling algorithm is preemptive but it depends on the implementation.

* Solaris is preemptive, Macintosh and windows are time sliced

Thread synchronization:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitor.

Each object in Java is associated with a monitor, which a thread can lock or unlock. Only ~~one~~ thread at a time may hold a lock on a monitor.

Java provides a very ~~handy~~ handy way of creating threads and synchronizing their task by using synchronized blocks.

Inter - thread communication

Inter - thread communication (co-operation) is all about allowing synchronized threads to communicate with each other.

Co-operation is a mechanism in which a thread is passed running in its critical section. and another thread is allowed to enter (or lock) in the same critical section to be executed
It is implemented by following methods of object class
* wait(), notifyAll(), notify()

PROGRAMS

1) DAEMON EXAMPLE:

CODE:

```
package java_prog;

public class DaemonExample extends Thread{
        public void run(){
         if(Thread.currentThread().isDaemon()) {
          System.out.println("This is Daemon  Thread");
         }
         else{
         System.out.println("This User Thread");
        }
        }
        public static void main(String[] args){
               DaemonExample thread1=new DaemonExample();
               DaemonExample thread2=new DaemonExample();
               DaemonExample thread3=new DaemonExample();

         thread1.setDaemon(true);

         thread1.start();
         thread2.start();
         thread3.start();
        }

       }
```

OUTPUT:

```
This is Daemon  Thread
This is User Thread
This is User Thread
```

2) Inter Thread Communication EXAMPLE:

CODE: ThreadInterComm.java

```
package java_prog;

class ThreadInterComm{
int amount=12000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw... \n");
System.out.println("Amount in account :"+this.amount);


if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
System.out.println("Going to withdraw an amount of " + amount +"\n");
this.amount-=amount;
System.out.println("withdraw completed...\n" );
System.out.println("Available amount in the a/c after withdrawal is
:"+this.amount+"\n");
}

synchronized void deposit(int amount){
System.out.println("Going to deposit of an amount of : "+amount+"\n");
this.amount+=amount;
System.out.println("Deposit completed and total amount in a/c : "+this.amount+"\n");
notify();
}
}
```

CODE: RunInterThreadComm.java

```
package java_prog;


class RunInterThreadComm{
public static void main(String args[]){
final ThreadInterComm c=new ThreadInterComm();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}
}
```

OUTPUT:

```
going to withdraw...

Amount in account :12000
Less balance; waiting for deposit...
Going to deposit of an amount of : 10000

Deposit completed and total amount in a/c : 22000

Going to withdraw an amount of 15000

withdraw completed...

Available amount in the a/c after withdrawal is :7000
```

3) Multi Extends Thread EXAMPLE:

CODE: MultiExtendsThread.java

```
package java_prog;

class MultiExtendsThread extends Thread {
          private Thread thread;
          private String threadName;

          MultiExtendsThread( String name) {
             threadName = name;
             System.out.println("Now Creating " +  threadName );
          }

          public void run() {
             System.out.println("Now Running " +  threadName );
             try {
                for(int i = 4; i > 0; i--) {
                   System.out.println("Thread: " + threadName + ", " + i);
                   Thread.sleep(50);
                }
             } catch (InterruptedException e) {
                System.out.println("Thread " +  threadName + " interrupted.");
             }
             System.out.println("Thread " +  threadName + " exiting.");
          }

          public void start () {
             System.out.println("Now Starting " +  threadName );
             if (thread == null) {
                thread = new Thread (this, threadName);
                thread.start ();
             }
          }
       }
```

CODE: RunMultiExtendsThread.java

```
package java_prog;

public class RunMultiExtendsThread {

          public static void main(String args[]) {
                MultiExtendsThread Thread1 = new MultiExtendsThread( "First Thread");
             Thread1.start();

             MultiExtendsThread Thread2 = new MultiExtendsThread( "Second Thread");
             Thread2.start();
          }
       }
```

OUTPUT:

```
Now Creating First Thread
Now Starting First Thread
Now Creating Second Thread
Now Starting Second Thread
Now Running First Thread
Thread: First Thread, 4
Now Running Second Thread
Thread: Second Thread, 4
Thread: Second Thread, 3
Thread: First Thread, 3
Thread: First Thread, 2
Thread: Second Thread, 2
Thread: First Thread, 1
Thread: Second Thread, 1
Thread First Thread exiting.

Thread Second Thread exiting.
```

4) Multi Runnable Thread EXAMPLE:

CODE: MultiRunnableThread.java

```
package java_prog;

class MultiRunnableThread implements Runnable {
String name;
Thread t;
 MultiRunnableThread (String threadname){
    name = threadname;
    t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start();
}



public void run() {
 try {
     for(int i = 5; i > 0; i--) {
     System.out.println(name + ": " + i);
      Thread.sleep(1000);
}
}catch (InterruptedException e) {
     System.out.println(name + "Interrupted");
}
     System.out.println(name + " exiting.");
}
}
```

CODE: RunMultiRunnableThread.java

```
package java_prog;

class RunMultiRunnableThread {
public static void main(String args[]) {
     new MultiRunnableThread("One");
     new MultiRunnableThread("Two");
     new MultiRunnableThread("Three");
try {
     Thread.sleep(10000);
} catch (InterruptedException e) {
```

```
      System.out.println("Main thread Interrupted");
}
      System.out.println("Main thread exiting.");
      }
}
```

OUTPUT:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
Two: 4
One: 4
Three: 4
Two: 3
Three: 3
One: 3
Three: 2
One: 2
Two: 2
Three: 1
Two: 1
One: 1
One exiting.
Three exiting.
Two exiting.
Main thread exiting.
```

5) SingleExtendsThread EXAMPLE:

CODE: SingleExtendsThread.java

```
package java_prog;

class SingleExtendsThread extends Thread{
    public static int myCount = 0;
    public void run(){
        System.out.println("Starting Child Thread...");
        while(SingleExtendsThread.myCount <= 10){
            try{
                System.out.println("Child Thread: "+(++SingleExtendsThread.myCount));
                Thread.sleep(100);
            } catch (InterruptedException iex) {
                System.out.println("Exception in thread: "+iex.getMessage());
            }
        }
        System.out.println("Exiting child thread");
    }
}
```

CODE: RunSingleExtendsThread.java

```
package java_prog;

public class RunSingleExtendsThread {
    public static void main(String a[]){
        System.out.println("Starting Main Thread...");
        SingleExtendsThread mst = new SingleExtendsThread();
        mst.start();
        while(SingleExtendsThread.myCount <= 10){
            try{
                System.out.println("Main Thread: "+(++SingleExtendsThread.myCount));
                Thread.sleep(100);
            } catch (InterruptedException iex){
                System.out.println("Exception in main thread: "+iex.getMessage());
            }
        }
        System.out.println("End of Main Thread...");
    }
}
```

OUTPUT:

```
Starting Main Thread...
Main Thread: 1
Starting Child Thread...
Child Thread: 2
Main Thread: 3
Child Thread: 3
Main Thread: 4
```

```
Child Thread: 4
Main Thread: 5
Child Thread: 6
Child Thread: 7
Main Thread: 8
Main Thread: 9
Child Thread: 9
Main Thread: 10
Child Thread: 11
End of Main Thread...
Exiting child thread
```

6) SingleRunnableThread EXAMPLE:

CODE: SingleRunnableThread.java

```
package java_prog;

class SingleRunnableThread implements Runnable{

    public static int myCount = 0;
    public SingleRunnableThread(){

    }
    public void run() {
        while(SingleRunnableThread.myCount <= 10){
              System.out.println("Starting Child Thread...");
            try{
                System.out.println("Child Thread:
"+(++SingleRunnableThread.myCount));
                Thread.sleep(100);
            } catch (InterruptedException iex) {
                System.out.println("Exception in thread: "+iex.getMessage());
            }
        }
        System.out.println("Exiting child thread..");
    }
}
```

CODE: RunSingleRunnable.java

```
package java_prog;

public class RunSingleRunnable {
        public static void main(String a[]){
               System.out.println("Starting Main Thread...");
               SingleRunnableThread mrt = new SingleRunnableThread();
               Thread t = new Thread(mrt);
               t.start();
               while(SingleRunnableThread.myCount <= 10){
                   try{
                       System.out.println("Main Thread:
"+(++SingleRunnableThread.myCount));
                       Thread.sleep(100);
                   } catch (InterruptedException iex){
                       System.out.println("Exception in main thread:
"+iex.getMessage());
                   }
               }
               System.out.println("End of Main Thread...");
           }

}
```

OUTPUT:

```
Starting Main Thread...
Main Thread: 1
Starting Child Thread...
Child Thread: 2
Starting Child Thread...
Child Thread: 4
Main Thread: 3
Main Thread: 5
Starting Child Thread...
Child Thread: 6
Starting Child Thread...
Child Thread: 8
Main Thread: 7
Starting Child Thread...
Main Thread: 9
Child Thread: 10
Starting Child Thread...
Child Thread: 12
Main Thread: 11
End of Main Thread...
Exiting child thread..
```

6) ThreadSync EXAMPLE:

CODE: ThreadSync.java

```
package java_prog;

import java.util.*;
class ThreadSync
{
   synchronized int locking     (int a, int b){return a + b;}
    int              not_locking (int a, int b){return a + b;}

    private static final int ITERATIONS = 1000000;
    static public void main(String[] args)
    {
       ThreadSync tester = new ThreadSync();
         double start = new Date().getTime();
      for(long i = ITERATIONS; --i >= 0 ;)
            tester.locking(0,0);
        double end = new Date().getTime();
        double locking_time = end - start;
        start = new Date().getTime();
      for(long i = ITERATIONS; --i >= 0 ;)
            tester.not_locking(0,0);
        end = new Date().getTime();
        double not_locking_time = end - start;
        double time_in_synchronization = locking_time - not_locking_time;
        System.out.println( "Time lost to synchronization (millis.): "
                        + time_in_synchronization );
        System.out.println( "Locking overhead per call: "
                        + (time_in_synchronization / ITERATIONS) );
        System.out.println(
            not_locking_time/locking_time * 100.0 + "% increase" );
    }
}
```

OUTPUT:

```
Time lost to synchronization (millis.): 28.0
Locking overhead per call: 2.8E-5
12.5% increase
```