MODULE IV
MULTITHREADED PROGRAMMING

STRUCTURE

LEARNING OBJECTIVES

- To understand the Java thread model.
- To Learn how to create multiple threads and to learn how to use functions like isalive() and join().
- To understand AWT packages.

4.1 INTRODUCTION

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.
Threads can be created by using two mechanisms:
1. Extending the Thread class
2. Implementing the Runnable Interface

4.2 THE JAVA THREAD MODEL, THE MAIN THREAD, CREATING A THREAD, CREATING MULTIPLE THREAD, CREATING A THREAD, CREATING MULTIPLE THREADS, USING ISALIVE() AND JOIN()

The Java thread model- In this document we describe two common models for Java Threads in which the threads can share certain the data members of objects. A Java thread is an instantiation of the Thread class. A thread must be created from a *thread base* which is any object whatsoever which defines the run function. A thread contains a thread base and adds hidden control structures in order to permit the thread to run concurrently with other threads.

The Java language is, at its core, a threaded language. Every Java application runs at least two threads: the main thread started by the function:
public static void main(String args[]);
and a garbage collection thread which deallocates unused pointer memory as needed. Let's say, for definiteness, that the class Main is the one which defines the main function. To start this Java application, we would execute the following statement (after compilation):
java Main

This starts the main thread. For the purposes of this course, the actions done are usually these:

- perform initializations such as creating shared objects
- create other thread base(s) and threads
- start the threads
- terminate main immediately or wait for some signal to terminate

The Thread Base

The thread base is an instantiation of a class which either:

1. implements Runnable
2. extends Thread

In either case the thread base class must define the following function:
public void run();

In the former case, the base must define run in order to implement the Runnable interface. In the latter case, the Thread class itself serves as the base and provides a trivial definition of the run function. The extending class must override this trivial definition.
Once a thread t is created from a thread base it is started by executing the start() member function — t.start() — which does certain initializations and executes the run function in the thread base.
Model One: shared base.

Assume there is a class ThreadBase which implements the *Runnable* interface. The main function creates a single thread base as follows:
ThreadBase base = new ThreadBase();
The threads are then created from the same thread base:
Thread A = new Thread(base, "A"); // "A" is the name of thread A
Thread B = new Thread(base, "B"); // "B" is the name of thread B
and then started by:
A.start();
B.start();
A and B share all of base's data member variables. A and B have the same run function and so they are effectively identical except for the *names* given to them in the second parameter. Only data member variables are shared, not the local (stack) variables used by the run function

Model Two: Separate base

Assume there are three classes in addition to the Main class:
Shared
ThreadA extends Thread
ThreadB extends Thread
The main function creates a single object whose data members will be shared by the threads:
Shared s = new Shared();
Threads are created as follows
Thread A = new ThreadA(s);
Thread B = new ThreadB(s);
and then started as before. Threads A and B created in this way have distinct thread bases. Actually, ThreadA and ThreadB could be the same class, the fact remains that A and B have distinct thread bases.
Threads **A** and **B** intend to share the object s by virtue of s being passed through the constructor (by pointer-by-value). In particular, the ThreadA class will typically look something like this:

```
class ThreadA extends Thread
{
```

```
  ThreadA( Shared s ) { this.s = s; }  // set member = parameter

  private Shared s;

  ...
}
```

in which case the data member s in ThreadA points to the original object s.

Additional distinctions between Models

Model One makes an explicit distinction between the thread base and the thread, whereas in Model Two the thread base and the thread are effectively merged.

In Model One the threads share the member variables whereas in Model Two, only the object pointed to (not the variables) are shared. In particular, Model Two cannot share variables of the primitive types int, boolean, etc. One would have to use the associated wrapper classes Integer, Float, etc.

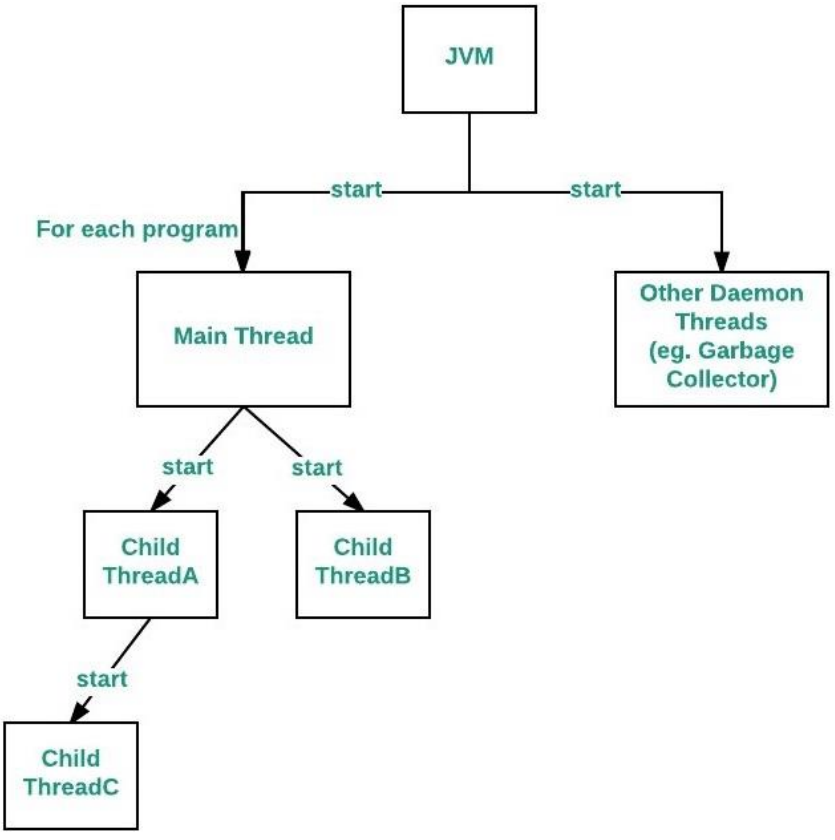
Model Two is generally simpler to use when threads execute different operations such as in most of the classical coordination problems:

The main thread- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins.

Properties:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions

Flow diagram:



How to control Main thread

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method *currentThread( )* which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

```
/ Java program to control the Main Thread
public class Test extends Thread
{
   public static void main(String[] args)
   {
      // getting reference to Main thread
      Thread t = Thread.currentThread();

      // getting name of Main thread
      System.out.println("Current thread: " + t.getName());

      // changing the name of Main thread
      t.setName("Geeks");
      System.out.println("After name change: " + t.getName());

      // getting priority of Main thread
      System.out.println("Main thread priority: "+ t.getPriority());

      // setting priority of Main thread to MAX(10)
      t.setPriority(MAX_PRIORITY);

      System.out.println("Main thread new priority: "+ t.getPriority());


      for (int i = 0; i < 5; i++)
      {
         System.out.println("Main thread");
      }

      // Main thread creating a child thread
      ChildThread ct = new ChildThread();

      // getting priority of child thread
      // which will be inherited from Main thread
      // as it is created by Main thread
      System.out.println("Child thread priority: "+ ct.getPriority());

      // setting priority of Main thread to MIN(1)
      ct.setPriority(MIN_PRIORITY);

      System.out.println("Child thread new priority: "+ ct.getPriority());

      // starting child thread
      ct.start();
   }
}
```

```
// Child Thread class
class ChildThread extends Thread
{
   @Override
   public void run()
   {
      for (int i = 0; i < 5; i++)
      {
         System.out.println("Child thread");
      }
   }
}
```

Output:

```
Current thread: main
After name change: Geeks
Main thread priority: 5
Main thread new priority: 10
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread priority: 10
Child thread new priority: 1
Child thread
Child thread
Child thread
Child thread
Child thread
```

Creating a thread- There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class ext implements Runnable interface.

Commonly used Constructors of Thread class:

Thread()
Thread(String name)
Thread(Runnable r)
Thread(Runnable r,String name)

Commonly used methods of Thread class:

public void run(): is used to perform action for a thread.
public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
public void join(): waits for a thread to die.
public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

public int getPriority(): returns the priority of the thread.
public int setPriority(int priority): changes the priority of the thread.
public String getName(): returns the name of the thread.
public void setName(String name): changes the name of the thread.
public Thread currentThread(): returns the reference of currently executing thread.
public int getId(): returns the id of the thread.
public Thread.State getState(): returns the state of the thread.
public boolean isAlive(): tests if the thread is alive.
public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspend(): is used to suspend the thread(depricated).
public void resume(): is used to resume the suspended thread(depricated).
public void stop(): is used to stop the thread(depricated).
public boolean isDaemon(): tests if the thread is a daemon thread.
public void setDaemon(boolean b): marks the thread as daemon or user thread.
public void interrupt(): interrupts the thread.
public boolean isInterrupted(): tests if the thread has been interrupted.
public static boolean interrupted(): tests if the current thread has been interrupted.

Creating multiple thread- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.
Threads can be created by using two mechanisms :
1. Extending the Thread class
2. Implementing the Runnable Interface

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread
{
   public void run()
   {
      try
      {
         // Displaying the thread that is running
         System.out.println ("Thread " +
               Thread.currentThread().getId() +
               " is running");

      }
      catch (Exception e)
      {
         // Throwing an exception
         System.out.println ("Exception is caught");
      }
   }
}

// Main Class
public class Multithread
{
```

```
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

filter_none
edit
play_arrow

brightness_4

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
```

```
}

// Main Class
class Multithread
{
   public static void main(String[] args)
   {
      int n = 8; // Number of threads
      for (int i=0; i<n; i++)
      {
         Thread object = new Thread(new MultithreadingDemo());
         object.start();
      }
   }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

Creating multiple threads Using isalive() and Join()-Thread provides a means by which you can answer this question. Java multi-threading provides two ways to find that

isAlive() : It tests if this thread is alive. A thread is alive if it has been started and has not yet died. There is a transitional period from when a thread is running to when a thread is not running. After the run() method returns, there is a short period of time before the thread stops. If we want to know if the start method of the thread has been called or if thread has been terminated, we must use isAlive() method. This method is used to find out if a thread has actually been started and has yet not terminated.

General Syntax :
final boolean isAlive( )

Return Value: returns true if the thread upon which it is called is still running. It returns false otherwise.

join() : When the join() method is called, the current thread will simply wait until the thread it is joining with is no longer alive.Or we can say the method that you will more commonly use to wait for a thread to finish is called join( ). This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Syntax :
final void join( ) throws InterruptedException
Here is an improved version of the preceding example that uses join( ) to ensure that the main thread is the last to stop. It also demonstrates the isAlive( ) method.
filter_none
edit
play_arrow

brightness_4
// Java program to illustrate
// isAlive()

```
public class oneThread extends Thread {
   public void run()
   {
      System.out.println("geeks ");
      try {
         Thread.sleep(300);
      }
      catch (InterruptedException ie) {
      }
      System.out.println("forgeeks ");
   }
   public static void main(String[] args)
   {
      oneThread c1 = new oneThread();
      oneThread c2 = new oneThread();
      c1.start();
      c2.start();
      System.out.println(c1.isAlive());
      System.out.println(c2.isAlive());
   }
}
```

Output:

geeks
true
true
geeks
forgeeks
forgeeks
Example of thread without join()

filter_none
edit
play_arrow

brightness_4
// Java program to illustrate
// thread without join()

```
public class oneThread extends Thread {
    public void run()
    {
        System.out.println("geeks ");
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException ie) {
        }
        System.out.println("forgeeks ");
    }
    public static void main(String[] args)
    {
        oneThread c1 = new oneThread();
        oneThread c2 = new oneThread();
        c1.start();
        c2.start();
    }
}
```

Output:

```
geeks
geeks
forgeeks
forgeeks
```

Example of thread with join()

filter_none
edit
play_arrow

brightness_4

```
// Java program to illustrate

public class oneThread extends Thread {
    public void run()
    {
        System.out.println("geeks ");
        try {
            Thread.sleep(300);
        }
        catch (InterruptedException ie) {
        }
        System.out.println("forgeeks ");
    }
    public static void main(String[] args)
    {
        oneThread c1 = new oneThread();
        oneThread c2 = new oneThread();
        c1.start();
```

```
        try {
            c1.join(); // Waiting for c1 to finish
        }
        catch (InterruptedException ie) {
        }

        c2.start();
    }
}
```

Output:

```
geeks
forgeeks
geeks
forgeeks
```

4.3 THREAD-PRIORITIES, SYNCHRONIZATION, INTER THREAD COMMUNICATION, SUSPENDING, RESUMING AND STOPPING THREADS, USING MULTI THREADING

Thread – Priorities- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Test it Now

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1

running thread priority is:1

Synchronization- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.
Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
  // Access shared variables and other
  // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

filter_none
edit
play_arrow

brightness_4

```
// A Java program to demonstrate working of
// synchronized.
import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
   public void send(String msg)
   {
      System.out.println("Sending\t"  + msg );
      try
      {
         Thread.sleep(1000);
      }
      catch (Exception e)
```

```
        {
            System.out.println("Thread  interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
    private String msg;
    Sender  sender;

    // Recieves a message object and a string
    // message to be sent
    ThreadedSend(String m,  Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
        // Only one thread can send a message
        // at a time.
        synchronized(sender)
        {
            // synchronizing the snd object
            sender.send(msg);
        }
    }
}

// Driver class
class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );

        // Start two threads of ThreadedSend type
        S1.start();
        S2.start();

        // wait for threads to end
        try
        {
```

```
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

Output:

```
Sending     Hi
 Hi Sent
Sending     Bye
 Bye Sent
```

Inter thread communication- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.Cooperation (Inter-thread communication) is a mechanism in which a thread is paused 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()
- notify()
- notifyAll()

wait() method-Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method Description-public final void wait()throws InterruptedException waits until object is notified.

public final void wait(long timeout)throws InterruptedException waits for the specified amount of time.

notify() method-Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

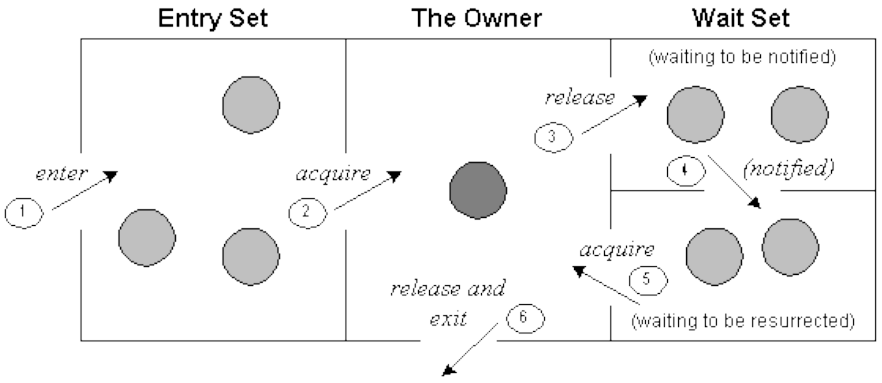
public final void notify()

notifyAll() method-Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

- Threads enter to acquire lock.
- Lock is acquired by on thread.
- Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
- If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object.

Suspending, resuming and stopping threads using multi threading

Program:

```
        /* ..... START ..... */

class MyThread implements Runnable {
          Thread thrd;
          boolean suspended;
          boolean stopped;

          MyThread(String name) {
            thrd = new Thread(this, name);
            suspended = false;
            stopped = false;
            thrd.start();
          }

          public void run() {
            try {
              for (int i = 1; i < 10; i++) {
                System.out.print(".");
                Thread.sleep(50);
                synchronized (this) {
                  while (suspended)
                    wait();
                  if (stopped)
                    break;
                }
```

```
            }
          } catch (InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
          }
          System.out.println("\n" + thrd.getName() + " exiting.");
        }

        synchronized void stop() {
          stopped = true;
          suspended = false;
          notify();
        }

        synchronized void suspend() {
          suspended = true;
        }

        synchronized void resume() {
          suspended = false;
          notify();
        }
      }

public class JavaThreadControl {
      public static void main(String args[]) throws Exception {
          MyThread mt = new MyThread("MyThread");
          Thread.sleep(100);
          mt.suspend();
          Thread.sleep(100);
          mt.resume();
          Thread.sleep(100);
          mt.suspend();
          Thread.sleep(100);
          mt.resume();
          Thread.sleep(100);

          mt.stop();
        }
}
          /* ..... END ..... */
```

Notes:
In multithreaded programming you can be suspended, resumed or stopped completely based on your requirements.
suspend() method puts a thread in suspended state and can be resumed using resume() method.
stop() method stops a thread completely.
resume() method resumes a thread which was suspended using suspend() method.

4.4 I/O BASICS, READING CONTROL INPUT, WRITING CONTROL OUTPUT, READING AND WRITING FILES

Basic Input & Output (I/O)

Programming simple I/O operations is easy, which involves only a few classes and methods. You could do it by looking at a few samples. Programming efficient, portable I/O is *extremely* difficult, especially if you have to deal with different character sets. This explains why there are so many I/O packages (nine in JDK 1.7)!

JDK has two sets of I/O packages:

1. the Standard I/O (in package java.io), introduced since JDK 1.0 for stream-based I/O, and
2. the New I/O (in packages java.nio), introduced in JDK 1.4, for more efficient buffer-based I/O.

JDK 1.5 introduces the formatted text-I/O via new classes *java.util.Scanner* and *Formatter*, and C-like *printf()* and *format()* methods for formatted output using format specifiers.

JDK 1.7 enhances supports for file I/O via the so-called NIO.2 (non-blocking I/O) in new package *java.nio.file* and its auxiliary packages. It also introduces a new *try*-with-resources syntax to simplify the coding of *close()* method.

Reading control input- In Java, there are three different ways for reading input from the user in the command line environment(console).

Using Buffered Reader Class

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

Advantages

- The input is buffered for efficient reading.

Drawback:

- The wrapping code is hard to remember.

Program:

```
// Java program to demonstrate BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
   public static void main(String[] args) throws IOException
   {
      //Enter data using BufferReader
      BufferedReader reader =
               new BufferedReader(new InputStreamReader(System.in));

      // Reading data using readLine
      String name = reader.readLine();

      // Printing the read line
      System.out.println(name);
   }
```

```
}
```
Input:
Geek
Output:
Geek
Note: To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

Using Scanner Class
This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line.

Advantages:
Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
Regular expressions can be used to find tokens.

Drawback:
- The reading methods are not synchronized

```
// Java program to demonstrate working of Scanner in Java
import java.util.Scanner;

class GetInputFromUser
{
   public static void main(String args[])
   {
      // Using Scanner for Getting Input from User
      Scanner in = new Scanner(System.in);

      String s = in.nextLine();
      System.out.println("You entered string "+s);

      int a = in.nextInt();
      System.out.println("You entered integer "+a);

      float b = in.nextFloat();
      System.out.println("You entered float "+b);
   }
}
```
Input:
GeeksforGeeks
12
3.4
Output:
You entered string GeeksforGeeks
You entered integer 12
You entered float 3.4

Using Console Class
It has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the

format string syntax can also be used (like System.out.printf()).

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

Does not work in non-interactive environment (such as in an IDE).

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
   public static void main(String[] args)
   {
      // Using Console to input data from user
      String name = System.console().readLine();

      System.out.println(name);
   }
}
```

Writing control output-

Console output is most easily accomplishedwith print() and println() methods, as described earlier. These methods are defined by the class PrintStream which is the type of object referenced by System.in. Even though System.out is a byte stream, using it for a simple program output is still acceptable.

Because the PrintStream is an output stream derived from the OutputStream, it also implements the low-level method write(). Thus, write() can be used to write to the console. The simplest form of write() defined by the PrintStream is shown below :

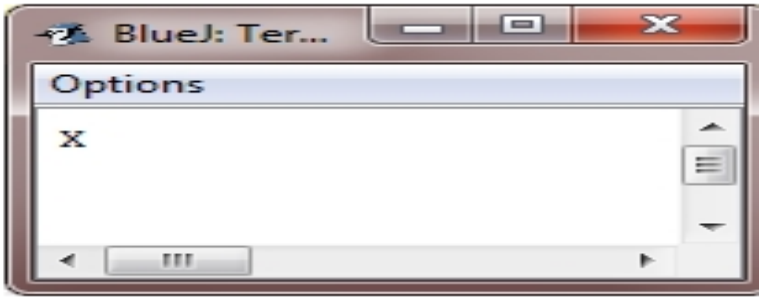
void write(int *byteval*)

This method writes the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written. Following is a short example that uses **write()** to output the character 'X' followed by a newline to the screen:

```
/* Java Program Example - Java Write Console Output
 * This program writes the character X followed by newline
 * This program demonstrates System.out.write()  */

 class WriteConsoleOutput
 {
    public static void main(String args[])
    {

       int y;

       y = 'X';

       System.out.write(y);
```

```
        System.out.write('\n');

    }
}
```

This Java program will produce the following output:



Reading and Writing files- One this page you can find a simple guide to reading and writing files in the Java programming language. The code examples here give you everything you need to read and write files right away, and if you're in a hurry, you can use them without needing to understanding in detail how they work. File handling in Java is frankly a bit of a pig's ear, but it's not too complicated once you understand a few basic ideas. The key things to remember are as follows.
You can read files using these classes:

- FileReader for text files in your system's default encoding (for example, files containing Western European characters on a Western European computer).
- FileInputStream for binary files and text files that contain 'weird' characters
  FileReader (for text files) should usually be wrapped in a BufferedFileReader. This saves up data so you can deal with it a line at a time or whatever instead of character by character (which usually isn't much use).
  If you want to write files, basically all the same stuff applies, except you'll deal with classes named FileWriter with BufferedFileWriter for text files, or FileOutputStream for binary files.

4.5 APPLET FUNDAMENTALS, THE AWT PACKAGE, AWT EVENT HANDLING CONCEPTS THE TRANSIENT AND VOLATILE MODIFIERS, USING INSTANCE OF USING ASSERT

Applet Fundamentals- An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.
There are some important differences between an applet and a standalone Java application, including the following −

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

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − 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.
- **start** − 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.
- **stop** − 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.
- **destroy** − 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.
- **paint** − 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.aw

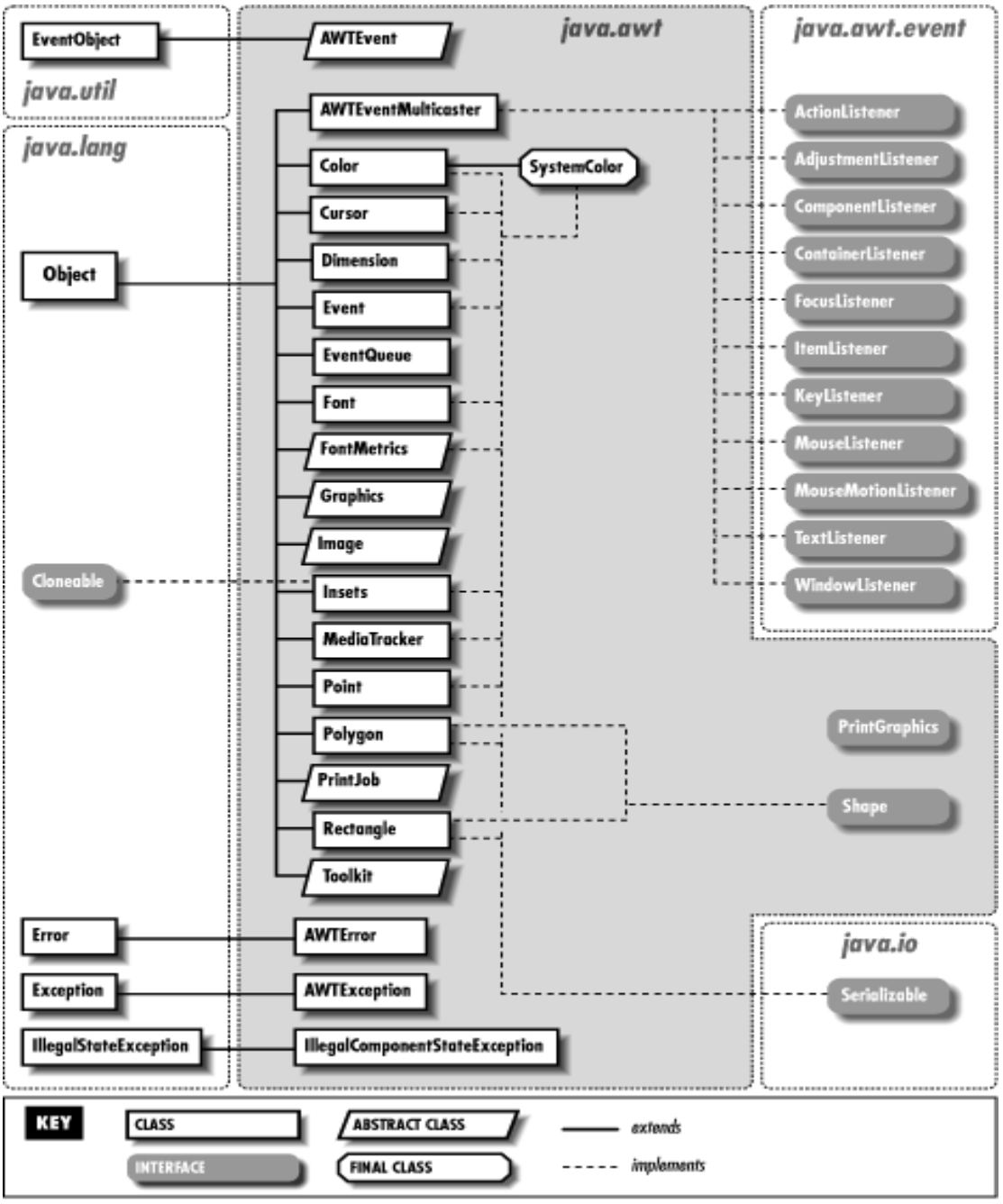
A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  }
}
```

The AWT package- The java.awt package is the Abstract Windowing Toolkit. The classes of this package may be roughly divided into three categories see Figure

- Graphics: These classes define colors, fonts, images, polygons, and so forth.
- Components: These classes are GUI (graphical user interface) components such as buttons, menus, lists, and dialog boxes.
- Layout Managers: These classes control the layout of components within their container objects.
- Note that separate packages, java.awt.datatransfer, java.awt.event, and java.awt.image, contain classes for cut-and-paste, event handling, and image m



Graphics, event, and exception classes of the java.awt package

In the first category of classes, Graphics is probably the most important. This class defines methods for doing line and text drawing and image painting. It relies on other classes such as Color, Font, Image, and Polygon. Image is itself an important class, used in many places in java.awt and throughout the related package java.awt.image. Event is another important class that describes a user or window system event that has occurred. In Java 1.1, Event is superseded by the AWTEvent class. Component and Menu Component are root classes in the second category of java.awt classes. Their subclasses are GUI components that can appear in interfaces and menus.

The Container class is one that contains components and arranges them visually. You add components to a container with the add() method and specify a layout manager for the container with the setLayout() method. There are three commonly used Container subclasses. Frame is a toplevel window that can contain a menu bar and have a custom cursor and an icon. Dialog is a dialog window. Panel is a container that does not have its own window--it is contained within some other

container.The third category of java.awt classes is the layout managers. The subclasses of Layout Manager are responsible for arranging the Component objects contained within a specified Container. GridBagLayout, Border Layout, and GridLayout are probably the most useful of these layout managers.

AWT Event handling- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:
Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event an then returns.
The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to that listener that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

The transient and volatile modifiers- transient is a variables modifier used in serialization. At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type. Transient keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of transient keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc. Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use transient keyword with private confidential fields of a class during serialization.

```
// A sample class that uses transient keyword to
// skip their serialization.
class Test implements Serializable
{
   // Making password transient for security
   private transient String password;

   // Making age transient as age is auto-
   // computable from DOB and current date.
   transient int age;
```

```
    // serialize other fields
    private String username, email;
    Date dob;

    // other code
}
```

Transient and static : Since static fields are not part of state of the object, there is no use/impact of using transient keyword with static variables. However there is no compilation error.

Transient and final : final variables are directly serialized by their values, so there is no use/impact of declaring final variable as transient. There is no compile-time error though.

```
// Java program to demonstrate transient keyword
// Filename Test.java
import java.io.*;
class Test implements Serializable
{
    // Normal variables
    int i = 10, j = 20;

    // Transient variables
    transient int k = 30;

    // Use of transient has no impact here
    transient static int l = 40;
    transient final int m = 50;

    public static void main(String[] args) throws Exception
    {
        Test input = new Test();

        // serialization
        FileOutputStream fos = new FileOutputStream("abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(input);

        // de-serialization
        FileInputStream fis = new FileInputStream("abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Test output = (Test)ois.readObject();
        System.out.println("i = " + output.i);
        System.out.println("j = " + output.j);
        System.out.println("k = " + output.k);
        System.out.println("l = " + output.l);
        System.out.println("m = " + output.m);
    }
}
```

Output :

```
i = 10
j = 20
k = 0
```

l = 40
m = 50

Volatile modifiers- The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory. Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.
Example

```
public class MyRunnable implements Runnable {
  private volatile boolean active;
  public void run() {
    active = true;
    while (active) {
    }
  }
  public void stop() {
    active = false;
  }
}
```

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you've skipped the pointers tutorial, you should go back and redo it. You must also be familiar with dynamic memory allocation and structures. Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

Assertions in Java
An assertion allows testing the correctness of any assumptions that have been made in the program. Assertion is achieved using the assert statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named AssertionError. It is mainly used for testing purposes during development.
The assert statement is used with a Boolean expression and can be written in two different ways.
First way :

```
assert expression;
```

Second way :

```
assert expression1 : expression2;
```

Example of Assertion:-

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;

class Test
{
   public static void main( String args[] )
   {
      int value = 15;
      assert value >= 20 : " Underweight";
      System.out.println("value is "+value);
   }
}
```

Output:
value is 15
After enabling assertions
Output:
Exception in thread "main" java.lang.AssertionError: Underweight
Enabling Assertions
By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:
java –ea Test
Or
java –enableassertions Test
Here, Test is the file name.

Disabling Assertions- The syntax for disabling assertions in java are:
java –da Test
Or
java –disableassertions Test
Here, Test is the file name.

Why to use Assertions

- Wherever a programmer wants to see if his/her assumptions are wrong or not.
- To make sure that an unreachable looking code is actually unreachable.

To make sure that assumptions written in comments are right.

```
if ((x & 1) == 1)
{  }
else // x must be even
{ assert (x % 2 == 0); }
```

- To make sure default switch case is not reached.
- To check object's state.
- In the beginning of the method
- After method invocation.

4.6 SUMMARY

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.
Threads can be created by using two mechanisms:
1. Extending the Thread class
2. Implementing the Runnable Interface

4.7 SELF ASSESSMENT QUESTION

1. Explain AWT Event handling
2. Explain Thread - Priorities
3. Explain resuming and stopping thread
4. Write notes on Applet

4.8 SUGGESTED READINGS

1. The complete reference Java –2: V Edition By Herbert Schildt Pub. TMH.
2. SAMS teach yourself Java – 2: 3rd Edition by Rogers Cedenhead and Leura Lemay Pub. Pearson Education
3. Java: A Beginner's Guide by Herbert Schildt
4. Java: Programming Basics for Absolute Beginners by Nathan Clark.