Core Java

Session5 By Saurabh Sharma

Agenda

Today we will cover the following two modules:

- Threads.
- Inner and Anonymous classes.

Module 1: Objectives

After completion of this module, you should be able to:

- Define Threads.
- Differentiate between Process and Thread .
- Explain types of Thread.
- Understand the life cycle of a thread.
- Explain java thread states and how to use it in Java.
- Create threads.
- Identify the thread priorities .
- Understand thread synchronization and inter-threaded communication .
- Explain garbage collection .

Defining a thread

- Sun defines a thread as a single sequential flow of control within a program.
- A threads is a sequence of lines of code that execute within a process.
- It is sometimes referred to as an execution context or a lightweight process.
- Thread based multitasking environments allow a single program to perform two or more tasks simultaneously.
- The thread in java is a realization of OS level thread. In other words, the thread in java is created using native methods which in turn create threads based on the OS.

Why threads ?

- Improved performance.
- Minimized system resource usage .
- Simultaneous access to multiple applications .
- Program structure simplification.
- Send & receive data on network.
- Read & write files to disk.
- Perform useful computation (editor, browser, game).

Process vs thread

Process	Thread
1. Executable program loaded in memory.	1. Sequentially executed stream of instructions.
2. Has own address space – variables & data structures (in memory)	2. Shares address space with other threads.
3. Communicate via operating system, files, network.	3. Has own execution context.
4. May contain multiple threads.	4. Multiple thread in process execute same program.

Multithreading

- Java provides built-in support for *multithreaded programming*. 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 multithreading is a specialized form of multitasking. Multithreading requires less overhead than multitasking processing.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- Disadvantage of multithreading:
  - Race condition.
  - Deadlock condition.
  - Lock starvation.

Multitasking vs Multithreading

Multitasking	Multithreading
❑ Refers to working with two or more programs concurrently.	❑ Refers to working with parts of one program.
❑ For example, working with MS-Word, Listening to mp3 song using Windows Media Player.	❑ For example, MS-Word program can check spelling, count line numbers, column number while writing in a document.

java.lang.Thread class

- Java SE provides 2 classes for working with thread called **Thread** and **ThreadGroup.**
- When a Java Virtual Machine starts up, that is when an application starts to execute, there is usually a single thread called **main**.
- The main threads continues to execute until :
  - **exit()** method is called.
  - all threads (non-daemon threads) have died.
- A deamon thread is a special type of thread that runs in background. When JVM exits, only remaining thread which will be running are daemon threads. By default all the threads that are created in java are non-daemon threads.

java.lang.Thread class (continued)

- Thread Class Declaration:

  public class Thread extends Object implements Runnable { }
- Constructor of Thread class
  - public Thread();
  - public Thread(String name);
  - public Thread(Runnable r);
  - public Thread(Runnable r, String name);
- Frequently used methods of Thread class
  - **public void start(): -** It will starts a thread by calling the run() method.
  - **public final String getName() : -** Shows the name of the thread.
  - **public static Thread currentThread() : -** Returns a reference to the currently executing thread object.
  - **public void run() : -** The entry point into thread.

java.lang.Thread class (Continued)

- Frequently used methods of Thread class
  - **public final boolean isAlive(): -** Determines whether a thread is running.
  - **public static void sleep(long millis): -** Makes the thread to pause for a period of time in milliseconds.
  - **public final void join(): -** Pauses until the thread terminates.
  - **public static void yield(): -** Pauses the currently executing thread & allow other to execute.
  - **Public final int getPriority(int): -** Return the priority of thread. Priority is between 1 to 10.

Lifecycle of thread

- New: - Created by instantiating the Thread class.
  - **Thread t1=new Thread;**
- Running or Runnable: - By calling the start() method that in turn call run() method.
  - **t.start();**
- Not Runnable: - Calling wait() or sleep() method or blocked state.
  - **t.wait();**
  - **t.sleep(1000);**
  - **Blocked state: -** The programmer can make a running thread to become inactive temporarily for some period. In this period (when inactive after starting), the thread is said to be in **blocked state**. The blocked state thread, as inactive, is not eligible to processor time. This thread can be brought back to the runnable state at any time. A thread can go a number of times from runnable state to blocked state and vice versa in its life cycle.
- Dead or Terminate: - After the run() method completes execution
  - **public void run { …… }**

Lifecycle of thread (continued)

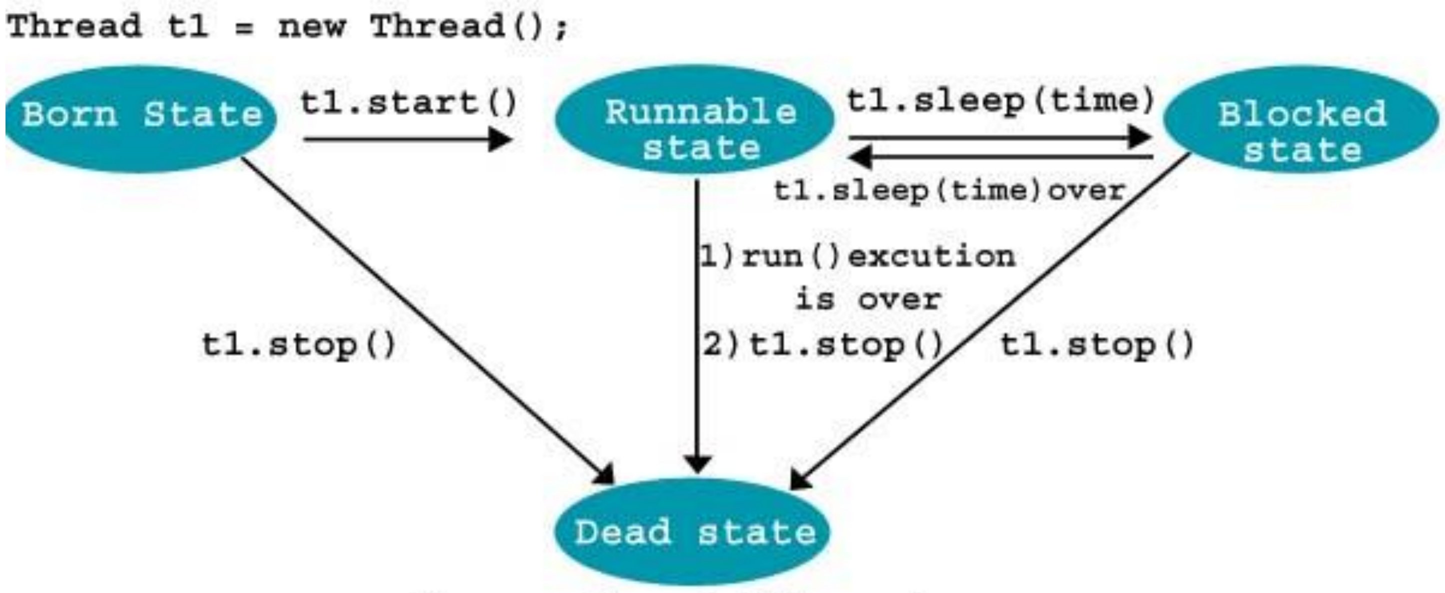


Figure: Thread life cycle

Lifecycle of thread (continued)

- Thread life cycle begins when a new thread instance is created.
- Then start() method on thread is called. Calling the start() method does not mean that the thread runs immediately. There may be many threads waiting to run. It is up to Thread scheduler of the OS to decide which thread will run first. We can’t control the scheduler from the java program! So when we have several threads (of same priority), we cannot determine which thread will run when.
- Also, just because a series of threads are started in a particular order doesn’t mean they’ll run in that order.
- When start() method is called a new thread of execution starts (with a new call stack).The thread moves from the new state to the ready state.
- When the thread gets a chance to execute, its run() method will be called.
- A thread may be blocked - waiting for a resource (like printer etc). Once the thread gets the resource, it again moves to ready state.

Creating threads

- Two ways of creating Thread in Java:
  - Extending from java.lang.Thread class.
    - Constructors that will be called in such case could be:
      - **Thread()**
      - **Thread(String name)**
  - Implementing the Runnable Interface.
    - Constructors that will be called in such case could be:
      - **Thread(Runnable target)**
      - **Thread(Runnable target, String name)**

Creating Threads by extending

1. class TestThread extends Thread {

2. public void run(){

/* code that is executed when

thread executes */

3. }}

- Creating the **TestThread** object

4. **TestThread t= new TestThread();**

5. **t.start(); //** Calls run() method of **TestThread** .

- We override the **run()** method and put the code that needs to be executed when the thread runs, in the **run()** method.
- To call run method we call **start()** !

Starting a thread

- Since **TestThread** inherits from the **Thread** class, **start()** method of the **Thread** class gets inherited into the **TestThread** class.
- The **start()** method of the **Thread** class creates OS level thread.
- After this it calls the **run()** method.
- Since the **run()** method is overriden , **run()** method of the **SimpleThread** is called.

Example: Creating Threads by extending

```
class TestThread extends Thread {
 private String name, msg;
   public MyThread(String name, String msg) {
       this.name = name;
       this.msg = msg;
   }
 public void run()    {
 System.out.println(name + " starts its execution");
 for (int i = 0; i < =5; i++)
{
 System.out.println(name + " says: " + msg+"for"+i+" time");
try {
      Thread.sleep(2000);
      }
catch (InterruptedException ie) {}
  }// End of For Loop
       System.out.println(name + " finished execution");
  }
    public static void main(String[] args)  {
         TestThread th1 = new TestThread("Nityo", "Hello Java");
           th1.start();
      }
}
```

Overridden Method

run() method is called. The New Thread moves to runnable state or to the running state depending upon the scheduler

Output of previous example

Output:

```
Nityo starts it’s execution
Nityo says: Hello Java for 1 time
Nityo says: Hello Java for 2 time
Nityo says: Hello Java for 3 time
Nityo says: Hello Java for 4 time
Nityo says: Hello Java for 5 time
Nityo finished execution
```

Note→ In this thread example ,we have used sleep method. we are passing some interval to the sleep method .After that interval thread will awake.

Problem with the first way – Using Runnable

- The way the thread was created (in the previous example) thread requires your class to inherit from **Thread** class.
- This means that this class cannot inherit from any other class.
- Another way to program threads is by implementing **Runnable** interface.
- **Runnable** interface has one method:
  - **public void run()**
- 2nd way to create thread is by
  - Creating a class that implements **Runnable** interface- that is overriding the **run().**
  - Creating a Thread class instance and passing **Runnable** instance. Recall the 2nd set of constructors of Thread.

Understanding working with Runnable

```
1. class TestThreadR implements Runnable{
2. public void run(){…}
3. }
//Creation of thread – using a constructor that expects a Runnable object
4. Thread t= new Thread(new TestThreadR() );
5. t.start(); // calls the run method of TestThreadR
```

- **Thread** class has a member called **target** which is of type **Runnable**.
- When the **Thread** constructor as above is invoked, the **target** member is assigned to the instance that is passed via constructor.
- When **start()** is called on the thread instance OS level thread gets created and then **run()** of Thread is called.
- **run()** of **Thread** in turn calls **run()** of target member.

Example: Code to creating Thread using Runnable

```
class TestThreadR implements Runnable {
  private String name, msg;
    public MyThread(String name, String msg) {
       this.name = name;
       this.msg = msg;
   }
 public void run()    {
 System.out.println(name + " starts its execution");
  for (int i = 0; i < =5; i++)
{
  System.out.println(name + " says: " + msg+” for “+i+”
time”);
 try {
      Thread.sleep(2000);
      }
catch (InterruptedException ie) {}
  }// End of For Loop
       System.out.println(name + " finished
       execution");
   }
    public static void main(String[] args)  {
         TestThreadR th1 = new
    TestThreadR(“Nityo", “Hello Java");
           Thread t1=new Thread(th1);
            t1.start();
      }
}
```

Overridden Method

Example: Code to creating Thread using Runnable

```
class TestThreadR implements Runnable {
  private String name, msg;
    public MyThread(String name, String msg) {
        this.name = name;
        this.msg = msg;
    }
  public void run()   {
  System.out.println(name + " starts its execution");
for (int i = 0; i < =5; i++)
 {
 System.out.println(name + " says: " + msg+" for "+i+"
time");
 try {
        Thread.sleep(2000);
        }
catch (InterruptedException ie) {}
   }// End of For Loop
        System.out.println(name + " finished
        execution");
  }
  public static void main(String[] args)  {
      TestThreadR th1 = new TestThreadR("Nityo",
      "Hello Java");
        Thread t1=new Thread(th1);
         t1.start();
    }
}
```

Overridden Method

A new Runnable object is created

Example: Code to creating Thread using Runnable

```
class TestThreadR implements Runnable {
 private String name, msg;
  public MyThread(String name, String msg) {
    this.name = name;
    this.msg = msg;
  }
 public void run()   {
 System.out.println(name + " starts its execution");
for (int i = 0; i < =5; i++)
 {
 System.out.println(name + " says: " + msg+" for "+i+"
time");
 try {
       Thread.sleep(2000);
      }
catch (InterruptedException ie) {}
   }// End of For Loop
      System.out.println(name + " finished
      execution");
 }
 public static void main(String[] args)  {
     TestThreadR th1 = new TestThreadR("Nityo",
     "Hello Java");
      Thread t1=new Thread(th1);
       t1.start();
   }
}
```

Overridden Method

A new Thread is created

Example: Code to creating Thread using Runnable

```
class TestThreadR implements Runnable {
 private String name, msg;
  public MyThread(String name, String msg) {
    this.name = name;
    this.msg = msg;
  }
 public void run()   {
 System.out.println(name + " starts its execution");
for (int i = 0; i < =5; i++)
 {
 System.out.println(name + " says: " + msg+" for "+i+"
time");
try {
     Thread.sleep(2000);
     }
catch (InterruptedException ie) {}
  }// End of For Loop
      System.out.println(name + " finished
      execution");
 }
 public static void main(String[] args)  {
     TestThreadR th1 = new TestThreadR("Nityo",
     "Hello Java");
      Thread t1=new Thread(th1);
       t1.start();
   }
}
```

Overridden Method

A newly created Thread is moved to running state by scheduler

Example: Code to creating Thread using Runnable

Output: It will also print the same output as previous example by extending Thread

```
Nityo starts it's execution
Nityo says: Hello Java for 1 time
Nityo says: Hello Java for 2 time
Nityo says: Hello Java for 3 time
Nityo says: Hello Java for 4 time
Nityo says: Hello Java for 5 time
Nityo finished execution
```

Note→ In this thread example ,we have used sleep method. we are passing some interval to the sleep method .After that interval thread will awake.

Naming Threads

- Every thread is given a name. If you don’t specify a name, a default name will be created.
- For example the main thread is named ‘main’.
- The default name of a user defined thread is ‘Thread-0’ for the first thread created, ‘Thread-1’ for the second and so on.
- To explicitly name the thread :
  - Use constructor:
    - **Thread(Runnable target, String name) or**
    - **Thread(String name)**
  - Or instance method :
    - **final void setName(String name)**
  - What is the method used to get the name? What is the method used to get the current thread?

Interruption

- A thread can be interrupted while it is sleeping or waiting. This can be done by calling **interrupt()** on it. In other words, when the thread is calling **sleep(), join() or wait()** methods, it can be interrupted .
- When **interrupt()** is called, an **InterruptedException** exception will be thrown.**.**So the catch handler block will be executed. **sleep(), join() or wait()** methods throw **InterruptedException**
- The **interrupt** status will be reset before the **InterruptedException** exception is thrown.
- Methods that can be used to test if the current thread has been interrupted:
  - **static boolean interrupted() : interrupted status** of the thread is cleared by this method.
  - **boolean isInterrupted(): interrupted status** of the thread is unaffected.
- *join() and wait() coming up!*

Example: interrupt

```java
import java.util.*;
public class TestInterrupt extends Thread {
  public void run() {
    while(true){
        try {
              System.out.println("Have a nice day");
                Thread.sleep(1000);
          } catch(InterruptedException e) { break; }
     }
  }
   public static void main(String args[]) {
       TestInterrupt t = new TestInterrupt ();
       t.start();
       Scanner scan = new Scanner(System.in);
       String s=scan.nextLine();
       if(s!=null)   t.interrupt();
} }
```

The code prints "Have a nice day" every 1 sec until user presses an a key then enter.
When user presses a key, an **InterruptedException** is thrown and catch handler is called. The **break** in catch handler makes the thread come out of while loop and hence **run()** method returns.

Example: Multithreading

```
import java.util.*;
public class MultiThreadE x implements Runnable{
  public void run() {
     for(int i=0;  i<5; i++){
        try {
              System.out.println(Thread.currentThread().
              getName());
                   Thread.sleep(1000);
           } catch(InterruptedException e) { }
     }   }
  public static void main(String args[]) {
      MultiThreadEx onj=new MultiThreadEx();
      Thread t=new Thread(obj, ”A”);
      Thread t1=new Thread(obj, ”B”);
      Thread t2=new Thread(obj, ”C”);
       t.start();
       t1.start();
       t2.start()
   }
}
```

Possible Output:

```
A
B
C
```

Prints 5 times but sequence is not guaranteed.

Point To Be Noted

The first thread to be executed in a multithreaded process is called the main thread.

The main thread is created automatically on the startup of java program execution.

Threads scheduling

- Scheduling: - The execution of multiple threads on a single CPU is called scheduling.
    - Determines which runnable threads to run.
    - Can be based on thread priority.
    - Part of OS or Java Virtual Machine (JVM).

- The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling.

join()

- When a thread calls **join( )** on another thread instance, the caller thread will wait till the called thread finishes execution.
- **final void join() throws InterruptedException**
- **join()** can also be specified with some timeout , in which case the thread waits at most milliseconds for this thread to die.
- If timeout specified as 0 means the thread will wait forever.
- **final void join(long millis) throws InterruptedException**
- **final void join(long millis, int nanos) throws InterruptedException**
- A thread waiting because of **join()** can also be interrupted. Hence the method throws a **InterruptedException.**

Example: join()

```
import java.util.*;
public class MultiThreadE x implements Runnable{
  public void run() {
     for(int i=0;  i<3; i++){
        try {
             System.out.println(Thread.currentThread().
             getName());
                 Thread.sleep(1000);
           } catch(InterruptedException e) { }
     }   }
   public static void main(String args[]) {
      MultiThreadEx onj=new MultiThreadEx();
      Thread t=new Thread(obj, ”A”);
      Thread t1=new Thread(obj, ”B”);
      Thread t2=new Thread(obj, ”C”);
       t.start();
      try { t.join(); }
     catch(InterruptedException e) { }
      t1.start()
      t2.start()
   }
}
```

Possible Output:

```
A
A
A
B
C
C
B
B
C
```

In this example **A** is guaranteed to print first but sequence is not guaranteed between **B and C.**

Thread priorities

- A thread is always associated with a priority based on the OS.
- For simplicity Java assigns priority as a number between 1 and 10, inclusive of 10. 10 is the highest priority and 1 is the lowest.
- This is the only way to influence the scheduler's decision as to the thread execution sequence to some extent.
- At any given time, the highest-priority thread will be chosen to run. However, this is not guaranteed. The thread scheduler may choose to run a lower-priority thread to avoid starvation.
- Threads that were created so far were created with a default priority of 5.
- The new threads inherit the priority from the thread that created it.

Methods for thread priorities

- Setting and getting priority :
  - **final void setPriority(int newPriority) throws IllegalArgumentException**
  - **final int getPriority()**
  - Note that **setPriority() throws IllegalArgumentException** if number passed to the method is not between 1 and 10 (inclusive of 10).
  - Also the priority cannot be greater than maximum permitted priority of the thread's thread group (even if the **newPriority** specifies a number greater than the thread's thread group priority).
- **static** constants to set priorities:
  - **Thread.MIN_PRIORITY (1)**
  - **Thread.NORM_PRIORITY (5)**→ default
  - **Thread.MAX_PRIORITY (10)**

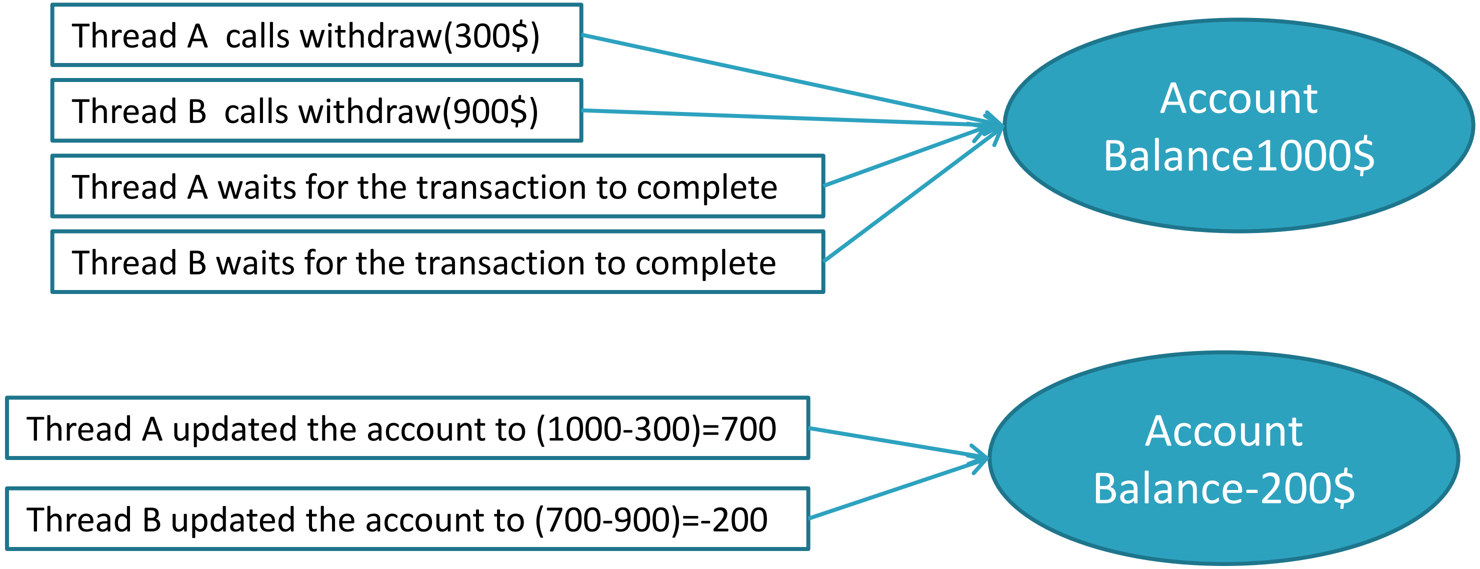
Selfish threads

- if a thread that executes with high priority and has long execution time does not voluntarily leave the CPU to give a chance for other threads with similar priority to execute, such a thread is called a selfish thread.
- It is recommended that threads with high priority (be unselfish).
- They must either call **Thread.sleep()** or **Thread. yield()** method.
  - **static void yield()**
- **yield()** method causes the currently executing thread object to temporarily pause and allow other threads to execute.

Why threads synchronization?

- To avoid Race condition!
  - Two or more threads access to same object and each call a method that modifies the state of the object.
- Example:
  - Let us say we have an Account class which has withdraw and deposit methods.
  - For each transaction (deposit or withdraw) a thread is created.
  - Let us visualize what happens when two people simultaneously withdraw from the same account object .
  - Let us assume that there is 1000$ in the account.

Why threads synchronization (Continued)?


Thread A calls withdraw(300$)
Thread B calls withdraw(900$)
Thread A waits for the transaction to complete
Thread B waits for the transaction to complete
Account Balance1000$
Thread A updated the account to (1000-300)=700
Thread B updated the account to (700-900)=-200
Account Balance-200$

Thread synchronization

- What is Synchronization?
  - Synchronization of threads ensures that if two or more threads need to access a shared resource then that resource is used by only one thread at a time.
- What is a Monitor?
  - A monitor is an object that can block threads and notify them when it is available.
  - The monitor controls the way in which synchronized methods access an object or class.
  - To enter an object’s monitor, you need to call a synchronized method.
- synchronized keyword: - It can be achieved at two levels
  - Method Level.
  - Object Level.

Thread synchronization (Continued)

- Synchronizing a method

```
public synchronized void withdraw( in dollar)
{
        // critical code goes here …
}
```

Only one thread will be inside the body of the withdraw(int dollar) method. The second call will be blocked until the first call returns or wait() is called inside the synchronized method.

Thread synchronization (Continued)

- Synchronizing an Object

```
  public void withdraw( in dollar)
  {
         synchronized(this)
         {
                        // critical code goes here …
         }
  }
```

In this case whole method is not synchronized, only we are synchronizing the part of method.

Note→ Because synchronization does hurt concurrency, we don't want to synchronize any more code than is necessary to protect our data for that purpose we use synchronize block.

Example: Code without synchronization

```
class Account {
    private int money;
    Account(int amt) {
    money=amt;       }
    void withdraw(int amt) {
          if(amt<money) {
             try {
                   Thread.sleep(1000);
                    money=money-amt;
                    }catch(Exception e){}
              System.out.println("Received "+ amt  +" by "
+     Thread.currentThread().getName());
              }
          else {
                  System.out.println("Sorry "+
Thread.currentThread().getName()+ "  Requested amt
("+ amt +") is not available.");
            }
            System.out.println("Balance "+ money);
  }}
class ThreadTest implements Runnable {
     Account a;
     int amt;
 public ThreadTest(Account a,String name,int amt)
 {
            this.a=a;
            this.amt=amt;
            new Thread(this,name).start();
 }
    public void run() {
              a.withdraw(amt); }
```

Example: Code without synchronization (Continued)

```
    public static void main(String str[])
    {
        Account lb= new Account(1000);
        ThreadTest t= new ThreadTest(lb,"A",300);
        ThreadTest t1=new ThreadTest(lb,"B",900);
    }
}
```

Output:

```
Received 300 by A

Balance 700

Received 900 by B

Balance -200
```

Solution to the Account problem

- Thread A acquires mutex for the Account object and calls withdraw(500).
- Thread B waits for A to release the lock.
- Thread A waits for the transaction to complete.
- Thread A updates the money to (1000-300)=700 and releases the lock.
- Thread B acquires lock for the object and calls withdraw(900).
- Thread B gets the message that it cannot withdraw the requested amount.
- Thread B releases the lock.

Correcting previous example using synchronized

- If an object is visible to more than one thread, all reads or writes to that object's non final attributes should be done through synchronized methods.
- Approach 1: Add synchronized keyword to withdraw and other critical methods of the Account object.

  synchronized void withdraw(int amt)

- Approach 2**:** Use **synchronized** statements by explicitly locking the object before calling critical methods of the Account object.
- **Example1:**

```
 public void run(){
      synchronized(a){
            a.withdraw(amt);
      }
  }
```

What about static method? Can they synchronized?

- How does **static synchronized** method work?
  - When they are invoked, since a **static** method is associated with a class, the thread acquires the monitor lock for the **Class** object associated with the class.
  - Thus access to class's static fields is controlled by this lock. Note that a lock on static method has no effect on any instances of that class.
- But what is **Class**?
  - Every class loaded in Java has a corresponding instance of java.lang.Class representing that class. This object contains all static members of the class X.
  - This Class object is in the method area (that is logically part of the heap).
  - The heap contains a pointer to the location of the object's class in the Method Area.

Deadlock

- A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects for example.
  - Thread 1 acquires intrinsic lock on resource 1.
  - Thread 2 acquires intrinsic lock resource 2.
  - Thread 1 waits for acquires intrinsic lock for resource 2 to be released.
  - Thread 2 waits for acquires intrinsic lock of resource 1 to be released.
  - Ends up in a DEADLOCK.

Points to be remember

- Synchronized methods of super class can be overridden to be unsynchronized.
- Constructors cannot be declared as synchronized because only the thread that creates an object should have access to it while it is being constructed.
- A non-static inner class can lock it's containing class using a synchronized block. We discuss about inner class later in this session.
- Methods in the interface cannot be declared as **synchronized**.
- The locking does not prevent threads from accessing **unsynchronized** methods.
- Synchronized methods are also called **thread-safe** methods.

Daemon threads

- So far the threads that we have been creating are called foreground threads. A program continues to execute as long as it has at least one foreground (non-daemon) thread that is alive.
- The daemon threads are also called service threads. They are used for background processes that will continue only as long as the active threads of the program are alive.
- Daemon threads cease to execute when there are no non-daemon threads alive because when VM detects that the only remaining threads are daemon threads, it exits.
- The threads we have seen so far are non-daemon.
- Example: garbage collector thread is a daemon thread that runs with a low priority.
- **final void setDaemon(boolean on) :** must be called before the thread is started.
- **boolean isDaemon()**

Example of Daemon Thread

```
class TestDaemon extends Thread {
public void run()
{
      while(true)    {
            try {
              System.out.println("Welcome To Nityo");
                Thread.sleep(1000);
            }
            catch(InterruptedException e) { break; }
      }
}
public static void main(String args[])
{
   TestDaemon d = new TestDaemon ();
   d.setDaemon(true);
   d.start();
   String str[]={"N","I","T","Y","O"};
try  {
   for(int i=0; i<str.length; i++)
   {
      System.out.print(str[i]+" ");
      Thread.sleep(1000);
   }
}
catch(InterruptedException e){}
System.out.println("End of main method");
} }
```

The code prints ““Welcome To Nityo“ for as long as main methods runs. The main method finishes execution after printing the last element of string array.

ThreadGroup

- **ThreadGroups** object have collection of threads.
- This helps in manipulating a group of threads all at a time instead of manipulating each of them individually . For instance all the threads in a collection can be started together.
- Every Java thread is a member of some thread group. When a thread is created, unless it is explicitly put in a thread group, the runtime system automatically places the new thread in the same group as the thread that created it.
- When a Java application first starts up, the Java runtime system creates a **ThreadGroup** named "**main**".

ThreadGroup Members

- Constructors
  - **ThreadGroup(String name)**
  - **ThreadGroup(ThreadGroup parent, String name)**
- **final ThreadGroup getThreadGroup()**

  Is a method of **Thread** class that returns the thread group the calling thread belongs to
- Methods to enumerate:
  - **int enumerate(Thread[] list)**
  - **int enumerate(Thread[] list, boolean recurse)**

ThreadGroup Members (continued)

- Methods with respect to the group
  - **int activeCount()**
  - **ThreadGroup getParent()**
  - **String getName()**
  - **boolean parentOf(ThreadGroup g)**
- Thread constructors with ThreadGroup
  - **Thread(ThreadGroup group, Runnable target)**
  - **Thread(ThreadGroup group, Runnable target, String name)**
  - **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**
  - **Thread(ThreadGroup group, String name)**

Methods to with respect to the threads as a group

- **int getMaxPriority()**
- **void setMaxPriority(int pri)**
- **void interrupt()**
- **void destroy()**
- **boolean isDaemon()**
- **void setDaemon(boolean daemon)**
- **void list()** (Prints information about this thread group to the standard output.)

Example: ThreadGroup

```
class ThreadGroupDemo{
	public static void main (String [] args){
			ThreadGroup tg = new ThreadGroup ("group 1");
			Thread t1 = new Thread (tg, "thread 1");
			Thread t2 = new Thread (tg, "thread 2");
			Thread t3 = new Thread (tg, "thread 3");
			tg = new ThreadGroup ("group 2");
			Thread t4 = new Thread (tg, "thread 4");
			tg = Thread.currentThread ().getThreadGroup ();
			int agc = tg.activeGroupCount ();
			System.out.println ("Active thread groups in " + tg.getName () +" thread group: " + agc);
			tg.list ();
}}
```

Output:

```
Active thread groups in main thread group: 2
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
    java.lang.ThreadGroup[name=group
1,maxpri=10]
    java.lang.ThreadGroup[name=group
2,maxpri=10]
```

Inter-thread communication

- Inter-thread communication is required when execution of one thread depends on another thread’s task.
- In such case, the second thread intimates or notifies the first thread when it has finished some task that the first thread is waiting for.
- The best suited situation to understand this is a producer-consumer problem.
- A producer thread produces something which consumer thread consumes. A producer and consumer thread can run independently.
- What we need to ensure however is that producer makes sure that it has produced enough for consumer to consume. If producer has not produced then consumer will have to wait till producer finishes.
- The various methods used in interthread communication are:
    - wait()
    - notify()
    - notifyAll()

wait()

- Consumer thread needs to check if the item to be consumed is there. Otherwise it has to wait. So a method is required to wait on the object where is item produced (object that is consumed is member of the object that is going to be consumed).
- Therefore some wait method has to be in the Object class.
  - **final void wait() throws InterruptedException**
  - **final void wait(long timeout) throws InterruptedException**
  - **public final void wait(long timeout, int nanos) throws InterruptedException**
- The first methods causes current thread to wait until either another thread invokes the **notify()** or the **notifyAll()** for this object,
- The second method waits for **notify()** or the **notifyAll()** for this object or waits for a specified amount of time which ever happens first.

notify()

- After the production of item, the Producer thread has to intimate the thread or threads which are waiting for the production. The production happens for a member of the an object.
- But the producer instead of directly notifying all the threads, just notifies the object. Since object has list of all waiting (consumer) threads with it (as a result of the wait calls), object awakens all the consumer threads.
  - **final void notify()**

    Wakes up a single thread that is waiting on this object's lock. The choice is arbitrary .
  - **final void notifyAll()**

    Wakes up all threads that are waiting on this object's lock.

Monitor

- Now one every important thing to bear in mind is that both producer and consumer thread have to own a monitor before producing or consuming.
- When **wait()** is called, the thread releases ownership of this monitor and waits until another thread notifies. Therefore the waiting threads has to wait until can re-obtain ownership of the monitor and resume execution.
- When the thread that calls **notify()** the lock is not released.
- Therefore the producer thread after producing calls **wait()** to relinquishes the lock on the object to allow consumer to consume.
- Similarly the consumer thread must notify the producer that it has finished consuming by calling **notify().** This awakens the producer thread which is on **wait().**

Exception thrown by wait and notify method

- **InterruptedException -** if another thread interrupted the current thread before or while the current thread was waiting for a notification.

- **IllegalMonitorStateException -** is thrown if these methods are not called from a synchronized context.

Example

```
class Calculator extends Thread {
      int total;
      public void run() {
          synchronized(this){
              for(int i=0; i<100; i++) {
                  total+=i;
              }
              notify();
          }
      }
}
class Reader extends Thread {
      Calculator c;
      public Reader(Calculator c) {
            this.c=c;
      }
public void run(){
    synchronized(c){
    try {
          System.out.println("Waiting for Calculation....");
          c.wait();
          }catch(InterruptedException e){}
          System.out.println("Total= "+c.total);
   } }
public static void main(String as[]) {
      Calculator cal=new Calculator();
       new Reader(cal).start();
       new Reader(cal).start();
        cal.start();  }}
```

Output:

```
Waiting for Calculation....
Waiting for Calculation....
Total= 4950
Total= 4950
```

Garbage collection

- Garbage collection is the feature of Java that helps to automatically destroy the objects created and release their memory for future reallocation.
- The various activities involved in garbage collection are:
  - Monitoring the objects used by a program and determining when they are not in use.
  - Destroying objects that are no more in use and reclaiming their resources, such as memory space.
  - The Java Virtual machine (JVM) acts as the garbage collector that keeps a track of the memory allocated to various objects and the objects being referenced.

Core Java: Session 5

Module 2: Inner class

Inner Class

- Like variables and methods, class can also be defined inside a class.
- An inner class is a class defined inside the scope of another class.
- Classes that were covered so far were top-level classes.
- The class inside which the inner class is defined is called outer class.
- Inner class can access even the private members of the outer class. Similarly outer class can also access the private members of inner class.
- Similar to inner class, inner interface can also be created.

Example: Inner Class

```
class Outer
{
        class  Inner{}
}
```

- Inner class instance has access to all member of the outer class(public , private & protected).
- Inner class can be access through live instance of outer class.
- The name of the inner class’s .class file name: **Outer$Inner.class** .

Types of Inner Class

- Member class
  - Static Inner Class/ Top-Level nested classes.
  - Non Static Inner Class.

- Local Inner Class.
- Anonymous Class.

- Non Static Inner Class, Local Inner Class, Anonymous Class are generally called inner class.
- Static inner class are considered to be top-level class.

Non static inner class

- Structure:

```
public class OuterClass{
   class InnerClass{..}
}
```

- Non static inner class object cannot be created without a outer class instance.
- The **private** fields and methods of the member classes are available to the enclosing class and other member classes.
- All the **private** fields and methods of the outer classes are also available to inner class.
- Non-static inner class cannot have **static** members.
- Other modifier applicable here are : - **abstract, final, public, protected, private.**

Example: Non static inner class instance

```java
class MyOuter
{
      private int x = 7;
      class MyInner
      {
            public void seeOuter()
            {
                  System.out.println("Outer x is " + x);
             }
       } // close inner class definition
       public static void main(String a[])
       {
              MyOuter mo = new MyOuter();
              MyOuter.MyInner inner = mo.new MyInner();
               inner.seeOuter();
       }

 }
```

- Outside outer class non-static inner class creation requires outer class instance also.
- There are 2 ways to do this.
  - If you don't need outer class instance , then create it like :

MyOuter.MyInner inner = new MyOuter.new MyInner();

  - If you need outer class instance or already have one, then create it like :

MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();

Outer class implicit reference in inner class

- Non-static inner class instance cannot exist without Outer class instance.
- This inner class has implicit reference to the outer class object using which it is created.
- Therefore no explicit reference required in inner class for the outer class.
- However, if outer class needs an inner class reference it has to create it explicitly.

Name conflict

- If the name of the members in Outer class and inner class are same, then how to refer to the name of the outer class member in the inner class?
- This can be done using Outer class name dot (.) this dot (.) member name.

```
class Outer {
    int i;
    class Inner {
        int i;
        void f()
        {
            i=10;
           Outer.this.i=9;
        }
    }
}
```

Static inner class

- Static nested classes referred to as top-level nested classes, or static inner classes, but they really aren’t inner classes at all, by the standard definition of an inner class.
- It is simply a non-inner (also called “top-level”) class scoped within another. So with static classes it’s really more about name-space resolution than about an implicit relationship between the two classes.
- A static nested class is simply a class that’s a static member of the enclosing class, as follows:

```
class BigOuter {
static class Nested { }
}
```

- The static modifier in this case says that the nested class is a static member of the outer class. That means it can be accessed, as with other static members, without having an instance of the outer class.
- It can access only all **static** members of the outer class.

Instantiating a static inner class

- The syntax for instantiating a static nested class is a little different from a normal inner class, and looks like this:

```
class BigOuter {
static class Nested { }
}
class Broom {
public static void main (String [] args) {
BigOuter.Nested n = new BigOuter.Nested(); //Use both class names
}
}
```

Method-Local Inner Classes

- An inner class that is defined inside a method is called local inner class (or method local inner class).
- A local inner class can be instantiated only by the method which defined it.
- Therefore no access specifier is applicable for the local inner class declaration. Only **abstract** and **final** modifiers are allowed.
- Also like other inner classes, local inner class can access all the members of the outer class including private members.
- Apart from the above, the local inner class can also access local variables which are **final**.
- **class OuterClass {**

```
 void someMethod(){
 class InnerClass{}
 }
```

Example:

- The following legal code shows how to instantiate and use a method-local inner class:

```
class MyOuter2 {
    private String x = "Outer2";
    void doStuff() {
        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
        }}
        MyInner mi = new MyInner(); // This line must come after the class
        mi.seeOuter(); }
    public static void main(String a[]) {
        MyOuter2 m1=new MyOuter2();
        m1.doStuff();
}} // close outer class
```

Example: using the local variables of the method the inner class is in.

```
class MyOuter2 {
    private String x = "Outer2";
    void doStuff() {
        String z = "local variable";
        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
                System.out.println("Local variable z is " + z); // Won't Compile!
            } // close inner class method
        } // close inner class definition
    } // close outer class method doStuff()
} //
```

Marking the local variable **z** as final fixes the problem:
final String z = "local variable"; // Now inner object can use it

Anonymous Inner Classes

- Inner class without a class name is an anonymous inner class.
- Allows creation of one time use object !
- Anonymous inner class can be created either inside a method or outside a method. It is implicitly **final**.
- No modifier is allowed anywhere in the class declaration.
- Also declaration cannot have an **implements** or **extends** clause.
- No constructors can be defined.
- An anonymous inner class is either inherited from an interface or from a class and so polymorphism is applicable. It cannot inherit from more than one class directly.

Syntax

- General way to create an anonymous inner class:

```
class OuterClass{
…
    SomeClassOrInterface obj = new SomeClassOrInterface()
    {
            //  overridden methods
    };
}
```

Note the semicolon here!

Example: Anonymous Inner Classes

```
class Popcorn {
        public void pop() {
        System.out.println("popcorn");
}}
class Food {
        Popcorn p = new Popcorn() {
                        public void pop()
                        {
                             System.out.println("anonymous popcorn");
                        }
        };
}
```

Let’s look at what’s in the preceding code:

- We define two classes, Popcorn and Food.
- Popcorn has one method, pop().
- Food has one instance variable, declared as type Popcorn.
- That’s it for Food. Food has no methods. And here’s the big thing to get:
- The Popcorn reference variable refers not to an instance of Popcorn, but to an instance of an anonymous (unnamed) **subclass** of Popcorn.
- Let’s look at just the anonymous class code:

```
Popcorn p = new Popcorn() {
   public void pop() {
      System.out.println("anonymous popcorn");
```

Inner class in interface and vice versa

- A class can be nested inside an interface. Though this is allowed in java, it is a bad practice to include implementation inside abstraction.
- A interface can be nested inside a class (or an interface). This is a very rarely used feature.

Questions