Java Day8

What is Multitasking?

- Multitasking is synonymous with process-based multitasking, whereas multithreading is synonymous with thread-based multitasking
- All modern operating systems support multitasking
- A process is an executing instance of a program
- Process-based multitasking is the feature by which the operating system runs two or more programs concurrently

Example

- You might have come across people doing multiple things at the same time: A person talking on the phone while having lunch or watching TV
- In a computer, you can run multiple programs at the same time. Like you can play a song using Media Player while typing a Word document

What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler
- A single program can perform two tasks using two threads
- Only one thread will be executing at any given point of time given a single-processor architecture

Examples

- Computer games are best examples of multithreading
- You might have seen that in most of the 'race' games, other cars or bikes will be competing with your car/bike. These are nothing but threads.

Examples

- Imagine that you need to paint your house. You can employ one painter who will take 10 days to complete the work or you can employ 10 painters who will finish the work in one day.
- In this case the 10 painters will be painting at the same time. That is 10 threads will be executing at the same time.

One thread

Multiple threads

Examples

- When you start typing in a MS Word document, you can see that the incorrectly spelled words are underlined with a red wavy line. Like:

Multiple threds in a progrm

- This is done by spell check feature of MS Word. The spell check is running in parallel while we type in a Word document. This is handled by a separate thread.

Multitasking Vs. Multithreading

Compared to multithreading, multitasking is characterized by the following:

- Each process requires its own separate address space
- Context switching from one process to another is a CPU-intensive task needing more time
- Inter-process communication between processes is again expensive as the communication mechanism has to span separate address spaces

These are the reasons why processes are referred to as heavyweight tasks

- Threads cost less in terms of processor overhead because of the following reasons:
  - Multiple threads in a program share the same address space and they are part of the same process
  - Switching from one thread to another is less CPU-intensive
  - Inter-thread communication, on the other hand, is less expensive as threads in a program communicate within the same address space

- Threads are therefore called lightweight processes

Extending Thread

- We can also create Threads by extending the Thread class:
  - Instantiate the class that *extends* Thread
  - This class must override *run()* method
  - The code that should run as a thread will be part of this *run()* method
  - We must call the *start()* method on this thread
  - *start()* in turn calls the thread's *run()* method

Extending Thread Example

A very simple demo for creating threads by extending Thread class:-

```
public class ThreadDemo1 extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
ThreadDemo1 threadDemo=new ThreadDemo1();
threadDemo.start();
 }
}
```

Extending Thread Example (Contd.).

One More Demo to show that thread is Running:-

```
public class ThreadDemo extends Thread{
public void run(){
for(int counter=1;counter<=100;counter++){
System.out.println("thread is running..."+counter);
}
}
public static void main(String args[]){
ThreadDemo threadDemo=new ThreadDemo();
threadDemo.start();
 }
}
```

The main Thread

- When a Java program starts executing:
  - the main thread begins running
  - the main thread is immediately created when **main()** commences execution

- Information about the main or any thread can be accessed by obtaining a reference to the thread using a public, static method in the **Thread** class called **currentThread()**

Obtaining Thread-Specific Information

```
public class ThreadInfo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread( );
    System.out.println("Current Thread :" + t);
    t.setName("Demo Thread");
    System.out.println("New name of the thread :"  +  t);
    try {
        Thread.sleep(1000);
    }
    catch (InterruptedException e) {
        System.out.println("Main Thread Interrupted");
    }
  }
}
```

Obtaining Thread-Specific Information (Contd.),

```
  public static void main(String args[]) {
    Thread t = Thread.currentThread( );
    System.out.println("Current Thread :" + t);
    t.setName("Demo Thread");
    System.out.println("New name of the thread :"  +  t);
    try {
        Thread.sleep(1000);
    }
    catch (InterruptedException e) {
        System.out.println("Main Thread Interrupted");
    }
  } }
```

Creating Threads: Implementing Runnable

- Thread can be created by creating a class which implements Runnable interface.

```
class DemoThread implements Runnable{}
```

- After defining the class that implements Runnable, we have to create an object of type Thread from within the object of that class. This thread will end when run( ) returns or terminates.
- This is mandatory because a thread object confers multithreaded functionality to the object from which it is created.
- Therefore, at the moment of thread creation, the thread object must know the reference of the object to which it has to confer multithreaded functionality. This point is borne out by one of the constructors of the Thread class.

4

The Thread class defines several constructors one of which is:

Thread(Runnable threadOb, String threadName)

- In this constructor, "threadOb" is an instance of a class implementing the Runnable interface and it ensures that the thread is associated with the run( ) method of the object implementing Runnable
- When the **run()** method is called, the thread is believed to be in execution.
- String threadName – we associate a name with this thread.
- Creating a thread does not mean that it will automatically start executing. A thread will not start running, until you call its **start( )** method. The **start( )** method in turn initiates a call to **run( ).**

Creating Threads: Implementing Runnable (Contd.).

A very simple demo for creating threads by implementing Runnable Interface:-

```
public class ThreadDemo implements Runnable {

public void run() {
System.out.println("thread is running...");
}

public static void main(String args[]) {
ThreadDemo threadDemo = new ThreadDemo();
Thread t1 = new Thread(threadDemo);
t1.start();
}
}
```

Output: thread is running…

6

One More Demo to show that thread is Running:-

```
public class ThreadDemo implements Runnable {

public void run() {
for(int counter=1;counter<=100;counter++){
System.out.println("thread is running..."+counter);
}
}

public static void main(String args[]) {
ThreadDemo threadDemo = new ThreadDemo();
Thread t1 = new Thread(threadDemo);
t1.start();
}

}
```

Output

threadis running...1

thread is running...2

thread is running...3

thread is running...4

thread is running...5

thread is running...6

thread is running...7

threadis running...8

...

thread is running...89

thread is running...90

thread is running...91

thread is running...92

threadis running...93

thread is running...94

thread is running...95

threadis running...96

thread is running...97

thread is running...98

threadis running...99

threadis running...100

Implementing Runnable or Extending Thread ?

- So, if the sole aim is to define an entry point for the thread by overriding the **run( )** method and not override any of the **Thread** class' other methods, it is recommended to implement the **Runnable** interface.

Creating Multiple Threads

- You can launch as many threads as your program needs
- The following example is a program spawning multiple threads:

```
public class ThreadDemo implements Runnable {

public void run() {
for(int counter=1;counter<=100;counter++){
System.out.println(Thread.currentThread().getName()+"thread is
running..."+counter);
}
}
```

Creating Multiple Threads (Contd.).

```
public static void main(String args[]) {
ThreadDemo threadDemo = new ThreadDemo();
Thread t1 = new Thread(threadDemo,"First");
Thread t2 = new Thread(threadDemo,"Second");
t1.start();
t2.start();
}

}
```

Output

```
Firstthread is running...1
Secondthread is running...1
Firstthreadis running...2
Secondthread is running...2
Secondthread is running...3
Secondthread is running...4
Secondthread is running...5
Secondthread is running...6
Secondthread is running...7
Secondthread is running...8
Secondthread is running...9
Secondthread is running...10
Secondthread is running...11
Secondthread is running...12
Secondthread is running...13
Secondthread is running...14
Secondthread is running...15
Secondthread is running...16
Secondthread is running...17
Secondthread is running...18
...
```

Thread Control Mechanisms

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

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

```
public class MultiThreadImpl {
public static void main(String args[]) {
DemoThread t1 = new DemoThread("One");
DemoThread t2 = new DemoThread("Two");
DemoThread t3 = new DemoThread("Three");
System.out.println("Thread One is alive: " +
t1.thread.isAlive());
System.out.println("Thread Two is alive: " +
t2.thread.isAlive());
System.out.println("Thread Three is alive: "  +
t1.thread.isAlive());


       try {
      System.out.println("Waiting for child  threads to
finish");
      t1.thread.join();
      t2.thread.join();
      t3.thread.join();
}
catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
}
```

Control Thread Execution (Contd.).

```
System.out.println("Thread One is alive: " +
t1.thread.isAlive());
System.out.println("Thread One is alive: " +
t1.thread.isAlive());
System.out.println("Thread One is alive: " +
t1.thread.isAlive());
System.out.println("Main thread exiting");
}
}
```

Control Thread Execution (Contd.).

Output:

```
New Thread: Thread[One,5,main]
New Thread: Thread[Two,5,main]
Child Thread: 5
New Thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for child  threads to finish
Child Thread: 5
Child Thread: 5
Child Thread: 4
Child Thread: 4
Child Thread: 4
```

Continued….

```
Child Thread: 3
Child Thread: 3
Child Thread: 3
Child Thread: 2
Child Thread: 2
Child Thread: 2
Child Thread: 1
Child Thread: 1
Child Thread: 1
ThreeExiting
OneExiting
TwoExiting
Thread One is alive: false
Thread One is alive: false
Thread One is alive: false
```

A Thought:

- When there are multiple threads running at the same time, how the CPU decides which thread should be given more time to execute and complete first?

3

Thread Priorities

- A thread priority decides:
  - The importance of a particular thread, as compared to the other threads
  - When to switch from one running thread to another
- The term that is used for switching from one thread to another is **context switch**
- Threads which have higher priority are usually executed in preference to threads that have lower priority
- When the thread scheduler has to pick up from several threads that are runnable, it will check the thread priority and will decide when a particular thread has to run
- The threads that have higher-priority usually get more CPU time as compared to lower-priority threads
- A higher priority thread can also preempt a lower priority thread
- Actually, threads of equal priority should evenly split the CPU time

Thread Priorities (Contd.).

- Every thread has a priority
- When a thread is created it inherits the priority of the thread that created it
- The methods for accessing and setting priority are as follows:

```
public final int getPriority( );
public final void setPriority (int level);
```

- When a new Java thread is created it has the same priority as the thread which created it
- Thread priority can be changed by the **setPriority()** method

```
thread1.setPriority(Thread.NORM_PRIORITY + 1);
thread2.setPriority(Thread.NORM_PRIORITY -1);
thread3.setPriority(Thread.MAX_PRIORITY - 1);

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

Example on Thread Priority

To demonstrate thread priority, we will use the same demo from the 'creating multiple threads' topic.
Modifying the program to include thread priority:

```
public class ThreadDemo implements Runnable {
public void run() {
for(int counter=1;counter<=100;counter++){
System.out.println(Thread.currentThread().getName()+"thread is running..."+counter);
}}
public static void main(String args[]) {
ThreadDemo threadDemo = new ThreadDemo();
Thread t1 = new Thread(threadDemo,"First");
t1.setPriority(Thread.MAX_PRIORITY);
Thread t2 = new Thread(threadDemo,"Second");
t2.setPriority(Thread.MIN_PRIORITY);
t1.start();
t2.start();
}}
```

Setting MAX priority to t1

Setting MIN priority to t2

9

Example on Thread Priority

Output:

```
Firstthread is running...1
Firstthread is running...2
Secondthread is running...1
Firstthread is running...3
Firstthread is running...4
Firstthread is running...5
Firstthread is running...6
Firstthread is running...7
Firstthread is running...8
Firstthread is running...9
Firstthread is running...10
Firstthread is running...11
Firstthread is running...12
Firstthread is running...13
Secondthread is running...2
Firstthread is running...14
Firstthread is running...15
Firstthread is running...16
Firstthread is running...17
Firstthread is running...18
Firstthread is running...19
Firstthread is running...20
```

We can see that the First thread is given more execution time than the Second thread since First thread is having MAX priority.

Deciding on a Context Switch

- A thread can voluntarily relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output
- All threads are examined and the highest-priority thread that is ready to run is given by the CPU
- A thread can be preempted by a higher priority thread
- A lower-priority thread that does not yield the processor is superseded, or preempted by a higher-priority thread

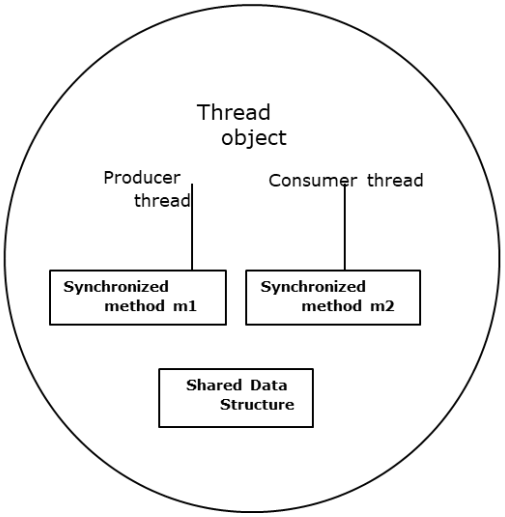
This is called preemptive multitasking.

- **When two threads with the same priority are competing for CPU time, threads are time-sliced in round-robin fashion in case of Windows like OSs**

Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- For example, imagine a Java application where one thread (which let us assume as Producer) writes data to a data structure, while a second thread (consider this as Consumer) reads data from the data structure

- This example use concurrent threads that share a common resource: a data structure.



Synchronization (Contd.).

- Suppose there is a producer thread in the threaded object that is writing into a data structure within the thread object using a method say M1.
- While the producer thread is in method M1 and in the process writing into the data structure, care must be taken to ensure that while data is in the process of being written to the data structure, no other thread, say a consumer thread must be allowed to read the data through some other method (say M2) of the thread object at the same time while the writing of data is on.
- The consumer thread should wait till the producer thread has finished writing into the data structure, i.e., till the producer thread returns from method M1.
- The moment the producer thread returns from method M1, the consumer thread should be allowed to access the data structure through method M2.

- The argument is moving towards a mechanism by which you are trying to ensure that no two threads end up accessing a shared data structure at the same time that might lead to corrupting the data in the data structure, and might also lead to unpredictable results. This mechanism will in fact serialize access to shared data by multiple threads, i.e., each thread lines up behind a running thread till the running thread has finished with its operation.

- The current thread operating on the shared data structure, must be granted mutually exclusive access to the data

- The current thread gets an exclusive lock on the shared data structure, or a **mutex**

- A **mutex** is a concurrency control mechanism used to ensure the integrity of a shared data structure

- Mutex is not assured, if, the methods of the object, accessed by competing threads are ordinary methods

- It might lead to a race condition when the competing threads will race each other to complete their operation

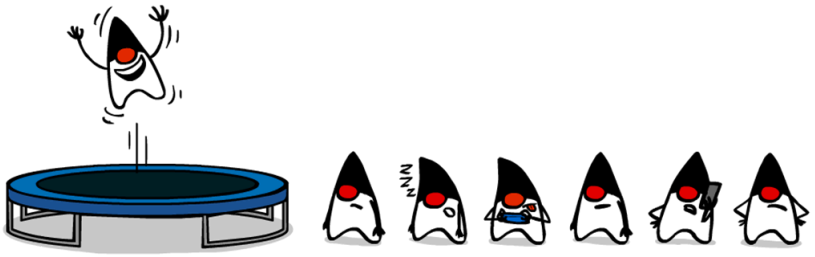
- A **race condition** can be prevented by defining the methods accessed by the competing threads as **synchronized**

- Synchronized methods are an elegant variation on a time-tested model of interprocess-synchronization: the ***monitor***

- The *monitor* is a thread control mechanism
- A monitor is an object, which is used as a concurrency control mechanism. It is used as a mutually exclusive lock(mutex).
- When a thread enters a monitor (synchronized method), all other threads, that are waiting for the monitor of same object, must wait until that thread exits the monitor

- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Example



Here when one Duke is using the trampoline, the other Dukes are waiting for their turn.
As we discussed in the last slide, when the first Duke enters a monitor, the remaining Duke's are waiting for the first one to complete.

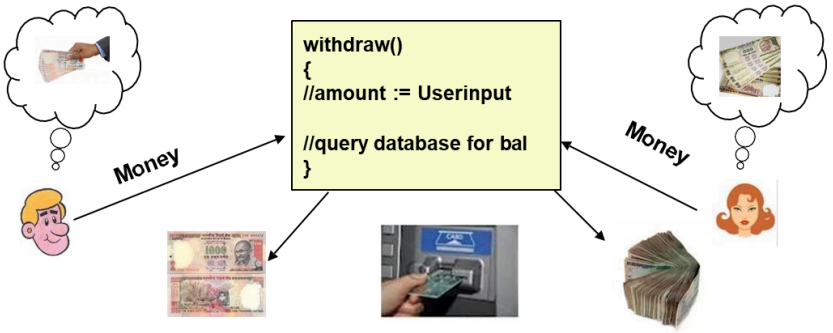
***Duke is Java's Mascot**

- Threads often need to share data.
- There is a need for a mechanism to ensure that the shared data will be used by only one thread at a time
- This mechanism is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore).

Using Synchronized Methods

- Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it
- If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object
- While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

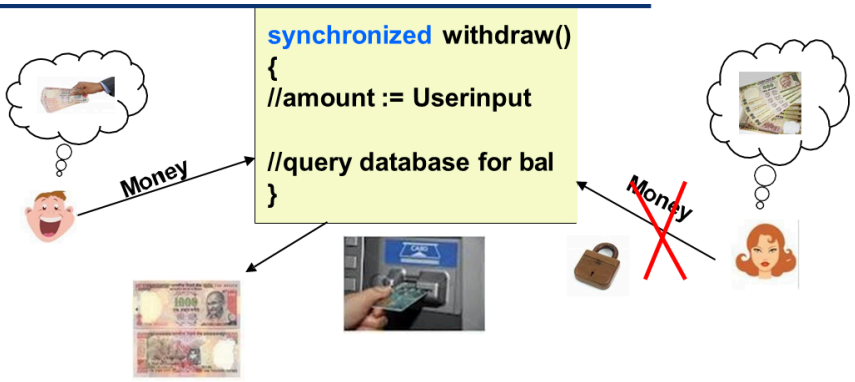
Using Synchronized Methods (Contd.).



John and Mary are withdrawing money from their joint account. They are withdrawing at the same **instance** from different ATMs.

What will be the concern here?

Each transaction occurs independently through different threads. Since both transactions are unaware of the other, this may lead to inconsistent state. How to avoid this situation?



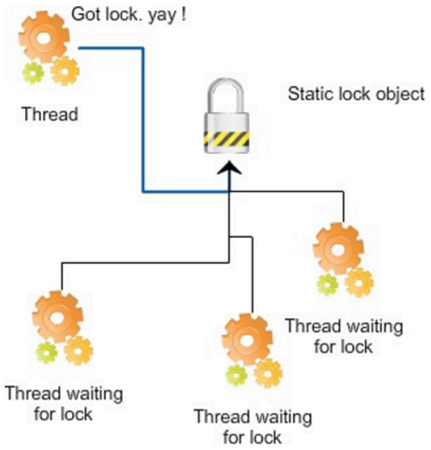
The concern shared in the previous slide can be easily handled using synchronization. When John wants to withdraw cash from his account, the thread that is responsible for handling this transaction gets an exclusive lock(monitor), which ensures that Mary cannot access this method concurrently. This mechanism comes in the form of synchronized method.

Synchronization

- Every object in Java has a lock
- Using ***synchronization*** enables the lock and allows only one thread to access that part of code
- Synchronization can be applied to:
  - A method
    ```
    public synchronized withdraw(){…}
    ```
  - A block of code
    ```
    synchronized (objectReference){…}
    ```
- Synchronized methods in subclasses use same locks as their superclasses

Synchronization

- When one thread acquires the lock on the shared resource, the other threads need to wait for the lock to be released.



Syntax for block of code:

```
public void run()
 {
    synchronized(obj)
    {
        obj.withdraw(500);
    }
  }
```

Implementing Synchronization-Example

```
class Account {
        int balance;
        public Account(){
                balance=5000;
        }
    public synchronized void withdraw(int bal){
                try{
                        Thread.sleep(1000);
                }
  catch(Exception ex){
      System.out.println("EXCEPTION OCCURED.."+ex);
  }
        balance= balance-bal;
        System.out.println("Balance remaining:::" + balance);
        }    }
```

Implementing Synchronization-Example

```
class C implements Runnable{
                    Account obj;
          public C(Account a){
          obj=a;
                    }
     public void run(){
                         obj.withdraw(500);
                    }                         }
class SynchEx{
                public static void main(String args[]){
                                   Account a1=new Account();
                                   C c1=new C(a1);
                                   Thread t1=new Thread(c1);
                                   Thread t2=new Thread(c1);
                                   t1.start();
                                   t2.start();
            }}
```