Core Java

objectives

- 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

Multitasking

- **Multitasking is performing two or more tasks at the same time. Nearly all OS are capable of it.**

- **Types**
  - Process Based
  - Thread Based

Process Based Multitasking

- **Means Running two or more programs concurrently.**
- **Here Programs are referred as process.**
- **E.g. you can run MS Word and MS Excel at the same time**

Thread Based Multitasking

- **Thread based multitasking means a single program perform two or more tasks.**
- **E.g. in word processing program can check the spelling of words in a document while you write the document**

Objective of Multitasking

- **To utilize the idle time of CPU.**
- **In many desk top application CPU remain idle.**
- **E.g You are using a word processor to write.**

For most part the CPU is idle until you enter a character from a keyboard or move the mouse

Objective of Multitasking(contd..)

- **For desktop application it is not too crucial.**
- **But it is crucial for programs that run in a networked environment need to make a CPU’s idle time productive.**

Process based VS Thread based Multitasking

- Each process requires its own address space in memory
- OS requires a significant amount of CPU time to switch from one process to another. (Context switching)
- Inter process communication requires additional resources

- Share same address space in memory because they are in the same program.
- Context switching does not consume much CPU time
- Inter thread communication is easy

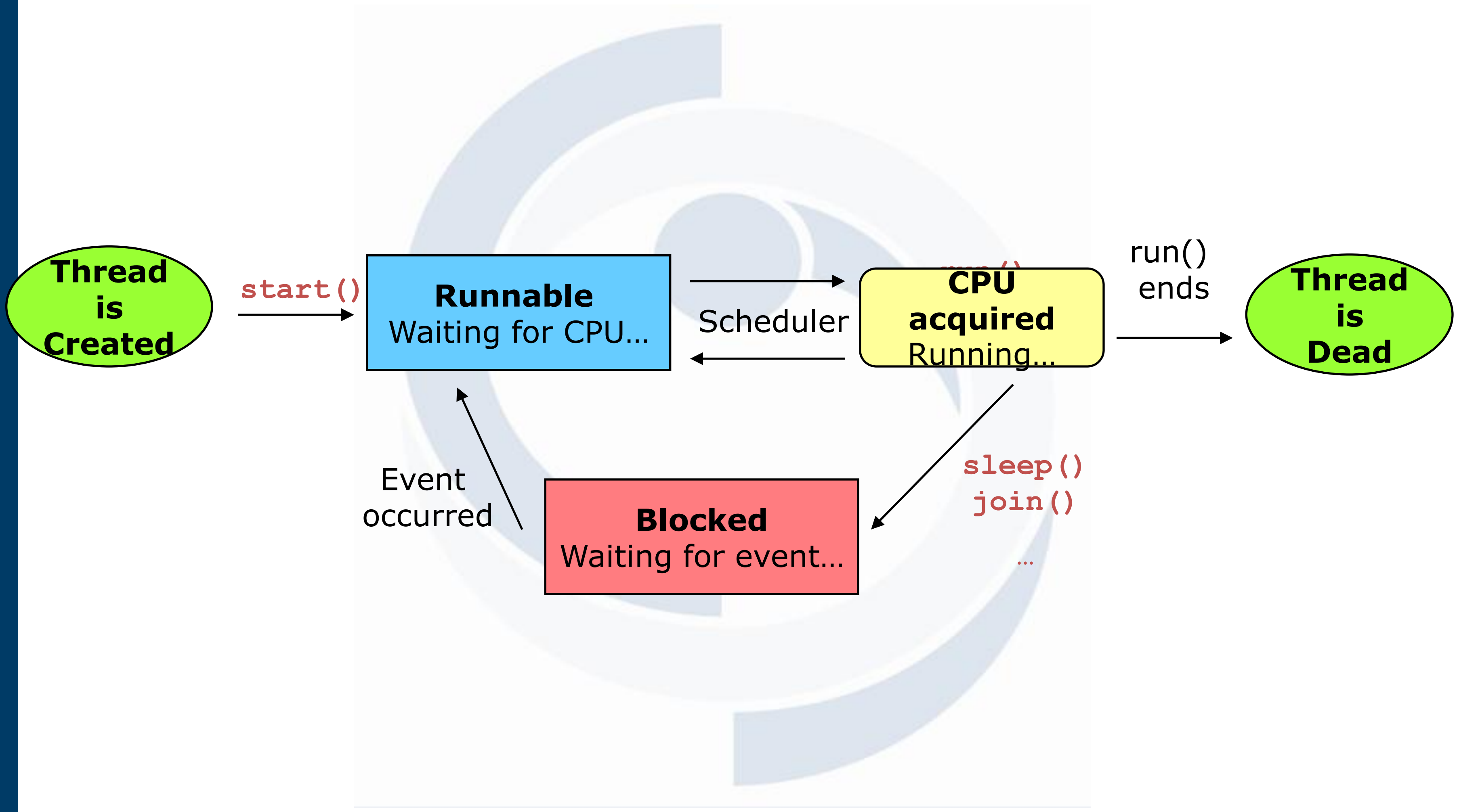
Thread

- **Thread is a light weight process**
- **Thread is a path of execution**
- **A thread is a part of a program that is running**
- **Thread based multitasking has multiple threads running at the same time.**
- **I.e multiple parts running concurrently**
- **E. g. Word processing – entering characters and spell check**

Multithreading in Java

- The Java run-time environment manages threads, unlike in process-based multitasking where the operating system manages switching between programs.
- All the class libraries are designed with multithreading in mind.

The Thread Life Cycle



Thread Priority

- All threads are not equal.
- Some threads are more important than other threads and are giving higher priority to resources such as the CPU.
- It is used in context switching (when to switch from one executing thread to another)

Rules for context switching

- **Rules for context switching**
  - A thread can voluntarily give control (by sleeping) to another thread. In doing so, control is turned over to the highest-priority thread.
  - A higher-priority thread can preempt a lower-priority thread for use of the CPU. The lower-priority thread is paused regardless of what it’s doing to give way to the higher-priority thread. Programmers call this *preemptive multitasking*

Thread Classes and the Runnable Interface

– There are two ways you can supply your application with threads:

  1. By implementing interface **`Runnable`**.
  2. By extending class java.lang.**`Thread`** (which actually implements **`Runnable`** by itself).

Thread Constructor

– Through one of the following constructors:

- `public Thread(Runnable target,String name)`
- `public Thread(String name)`
- `public Thread(ThreadGroup group,String name)`
- `public Thread(ThreadGroup group,Runnable target,String name)`

Methods in Thread Class

Method	**Description**
getName()	Returns the name of the thread.
getPriority()	Returns the priority of the thread.
isAlive()	Determines whether the thread is running.
join()	Pauses until the thread terminates.
run()	The entry point into the thread.
sleep()	Suspends a thread. This method enables you to specify the period
start()	Starts the thread.

Main Thread

- **Every Java program has one thread, even if you don't create any threads.**
-
- **This thread is called the *main* thread because it is the thread that executes when you start your program.**
- **The main thread spawns threads that you create. These are called *child* threads.**

Main Thread

- **You can control the main thread using methods in Thread class.**

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


Thread t = Thread.currentThread();


      System.out.println("Current thread: " + t);


      t.setName("Demo Thread");


      System.out.println("Renamed Thread: " + t);
   }
}
```

Thread creation by Runnable Interface

```
class MyThread implements Runnable {

  Thread t;

MyThread () {

    t = new Thread(this,"My thread");
     t.start();

}
public void run() {

    System.out.println("Child thread started");

    System.out.println("Child thread termina}
}

}
```

Thread creation by Runnable Interface

```
class Demo {
  public static void main (String args[]){
            new MyThread();

         System.out.println("Main thread started");

         System.out.println("Main thread terminated");

              }
    }
```

Thread creation by extending Thread

```
class MyThread extends Thread {
  MyThread(){
        super("My thread");
         start();
  }
  public void run() {
    System.out.println("Child thread started");
    System.out.println("Child thread terminated");
  }
}
```

Thread creation by extending Thread

```
class Demo {
  public static void main (String args[]) {
    new MyThread();
    System.out.println("Main thread started");
    System.out.println("Main thread terminated");
  }
}
```

Multithreading

```
class MyThread implements Runnable {
  String tName;
  Thread t;
  MyThread (String threadName) {
    tName = threadName;
    t = new Thread (this, tName);
    t.start();
  }
  public void run() {
    try {
       System.out.println("Thread: " + tName );
       Thread.sleep(2000);
    } catch (InterruptedException e ) {
      System.out.println("Exception: Thread "
              + tName + " interrupted");
    }
    System.out.println("Terminating thread: " + tName );
  }
}
```

Multithreading

```
class MultiThread1 {
   public static void main (String args []) {
      new MyThread ("1");
      new MyThread ("2");
      new MyThread ("3");
      new MyThread ("4");
      try {
         Thread.sleep (10000);
      } catch (InterruptedException e) {
        System.out.println(
                 "Exception: Thread main interrupted.");
      }
      System.out.println(
                  "Terminating thread: main thread.");
  }
}
```

Multithreading

- Typically, the main thread should be the last thread to finish in a program as it performs various shutdown actions.
- It is achieved by join() method
- The isAlive() method is used to check whether the thread is alive or not.

Thread Priority

- Priority is used as a guide for the operating system to determine which thread gets accesses to a resource such as the CPU.
- In reality, an operating system takes other factors into consideration.
- Typically, programmers have little or no control over those other factors.

Thread Priority

- **A priority is an integer from 1 to 10**
- **10 is the highest priority, referred to as the *maximum priority***
- **1 is the lowest priority, also known as the *minimum priority.***
- **The normal priority is 5, which is the default priority for each thread.**

Thread Priority

- **You can set the Thread Priority by method of Thread class**
- **setPriority(int priority)**
- **Thread.MIN_PRIORITY -> 1**
- **Thread.MAX_PRIOIRTY -> 10**
- **Thread.NORM_PRIOIRTY -> 5**

Synchronization

- **Multithreading occurs *asynchronously*, meaning one thread executes independently of the other threads.**
- **In this way, threads don’t depend on each other’s execution.**

Synchronization

- **Sometimes the execution of a thread is dependent on the execution of another thread.**
- **Let’s say you have two threads—one handles gathering login information, and the other validates a user’s ID and password.**
- **The login thread must wait for the validation thread to complete processing before it can tell the user whether or not the login is successful. Therefore, both threads must execute synchronously, not asynchronously**

Synchronization

- **The objective of synchronization is to ensure that multiple threads will not access a shared resource at the same time.**
- **A shared resource can, for example, be a file that a couple of threads are reading data from and some are writing data to a file.**
  - You don't wish the reading threads to access the file while the writing ones are in the middle of a writing procedure.
    - Failing to synchronize their activities will result in uncompleted or even corrupted data

Mutual Exclusion and Monitors

- **The ability of multiple threads to use a single data source without interfering with each other is achieved using a mechanism called mutual exclusion (or mutex).**
- **This goal is achieved using monitors**
  - All objects in Java have a monitor.
  - Only one thread can own a monitor at a given time.
  - In order to get hold of an object's monitor, a thread needs to arrive at the monitor's region.

Monitor Regions

- **There are actually 2 ways of creating the monitor region:**

  1. Synchronized method.

  2. Synchronized block:

Synchronized method

- **A thread enters a monitor whenever a method modified by the keyword synchronized is called.**

- **Another thread that calls the synchronized method is suspended until the first thread relinquishes the synchronized method.**

Synchronization

```
public class Callme {

synchronized public void show(String msg) {
                System.out.print(" [ "+msg);
          try {
                  Thread.sleep(2000);
              } catch (InterruptedException e) {
                System.out.println();
              }
               System.out.println("]");
           }
}
```

Synchronization

```
public class XThread extends Thread {
    Callme obj;
    String msg;
                public XThread(Callme obj,String msg) {
                   this.obj=obj;
                    this.msg=msg;
                    start();
}
 public void run() {
                                                obj.show(msg);
                                 }


}
```

Synchronization

```
public class SynchroDemo {

        public static void main(String[] args) {

                Callme C= new Callme();
                XThread x1=new XThread(C,"java");
                XThread x2=new XThread(C,"C++");
                XThread x3=new XThread(C,".Net");
        }

}
```

Synchronized block

- **The second technique for creating the monitored region is by using a synchronized block.**

- **No other synchronized blocks or methods will be executed on the locking object until this block is completed.**

Synchronized block

```
public class Callme {
        public void show(String msg)
            {
                synchronized (this) {
                        System.out.print(" [ "+msg);
                          try {
                                Thread.sleep(2000);
                                } catch (InterruptedException e) {

                                            e.printStackTrace();
                                }
                              System.out.println("]");
                        }
             }
}
```

Synchronized method And Block

- Synchronizing a method is the best way to restrict the use of a method one thread at a time.
- You can't use synchronize a method, when you use a class that is provided to you by a third party.
- In such cases, you don't have access to the definition of the class, which prevents you from using the synchronized keyword.
- An alternative to using the synchronized keyword is to use the synchronized statement.
- A synchronized statement contains a synchronized block, within which is placed objects and methods that are to be synchronized.

Communicating Between Threads

- **Sometimes threads have to coordinate their processing**
- **Therefore they need to communicate with each other during processing. E.g.Producer consumer**
- **Cooperation enables several threads to work together towards a shared goal.**
  - In order to do so, we use 3 methods of class **`Object`** - **`wait()`**, **`notify()`** and **`notifyAll()`**.

wait() notify() notifyAll()

- **These methods are called from within a synchronized method.**
  - wait() - tells a thread to relinquish a monitor and go into suspension.
  - notify() - tells a thread that is suspended by the wait()method to wake up again and regain control of the monitor.
  - The notifyAll() method wakes up all threads that are waiting for control of the monitor.

Wait and Notify Example

- **The following example demonstrates the cooperation mechanism.**
  - A **basket** object is used for passing information between two other objects, a **producer** (who fills the basket with products) and a **consumer** (who take these products out).
  - The basket can’t hold more than one product at a time.
  - The producer should produce products (one at a time), each will be consumed by the consumer (again, one at a time).
  - Naturally, both the producer and consumer run concurrently in two different threads

Inter Thread Communication

```
public class Basket {
   private int num;   private boolean flag=true;
            synchronized public int getNum() {
                        if(flag==true) {
                                    try {
                                             wait();
                                    } catch (InterruptedException e) {
                                                e.printStackTrace();  }   }
                        System.out.println(" got :"+num);
                        flag=true;
                        notify();
                        return num;
            }
```

Inter Thread Communication

```
synchronized public void setNum(int num) {
            if(flag==false)
            {
                  try {
                        wait();
                     } catch (InterruptedException e) {
                        e.printStackTrace(); }
            }
                this.num = num;
                System.out.println(" put :"+num);
                flag=false;
                notify();
          }
} //class Basket
```

Inter Thread Communication

```
public class Consumer extends Thread{
        Basket basket;
        public Consumer(Basket basket){

                this.basket=basket;
          }
        public void run(){
                for(int i=0;i<100;i++)  {
                        basket.getNum();
                        }
        }
}
```

Inter Thread Communication

```
public class Producer extends Thread{
        Basket basket;
            public Producer(Basket basket){
              this.basket=basket;
            }
            public void run(){

                    for(int i=0;i<100;i++) {
                            basket.setNum(i);
                    }

            }
}
```

Inter Thread Communication

```
public class ThreadCommunication {

        public static void main(String[] args) {
    Basket basket = new Basket();

    Producer producer= new Producer(basket);

    Consumer consumer= new Consumer(basket);

    producer.start();
    consumer.start();
        }

}
```

Deadlocks

- A deadlock is one kind of situation you really don't wish to encounter.
- It can happen when one thread (let’s call it threadA) synchronizes on a certain object (let’s say ObjectA) to perform some kind of task.
- Then, it goes to sleep and allows threadB to start executing.
- ThreadB synchronizes itself on ObjectB and ‘goes to sleep’.
- ThreadA wakes up and wishes to go on performing its duty, which, quite surprisingly, is located on ObjectB, which is locked by ThreadB.
- ThreadB takes a go at it, but is blocked by the locked ObjectA. .
- As we said earlier, you really don't want this to happen.