

Unit 7

Threads

Introduction to Threads

- ❖ A single thread is basically a lightweight and smallest unit of processing
- ❖ If a program has multiple thread, each thread can execute different parts of code in parallel with the main thread.
- ❖ Multithreaded applications execute two or more threads run concurrently for maximizing CPU utilization.
- ❖ There are two types of thread:
 - User thread
 - Daemon thread
- ❖ User thread or non-daemon thread are designed to do specific or complex task whereas daemon threads are used to perform supporting tasks.
- ❖ JVM terminates itself when all of the user threads finish their execution.

Introduction to Threads

- ❖ When a JVM starts up, there is a thread that calls the main method.
- ❖ This thread is called main

```
//checking how many threads are active  
System.out.println(Thread.activeCount());
```

```
//getting name of current thread  
System.out.println(Thread.currentThread().getName());
```

- ❖ Changing the thread name

```
//changing thread name  
Thread.currentThread().setName("MyThread");
```

- ❖ Checking if current thread is alive

```
System.out.println(Thread.currentThread().isAlive());
```

Introduction to Threads

❖ sleep(long milliseconds)

- This method will make the thread sleep hence the thread's execution will pause for milliseconds provided and after that, again the thread starts executing.

```
public static void main(String[] args) throws InterruptedException {  
    for(int i =5;i>=1;i--) {  
        System.out.println(i);  
        Thread.sleep(1000); //1000ms  
    }  
}
```

❖ yield()

- Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.

❖ join(long milliseconds)

- The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.

Creating a Thread

❖ There are two ways to create a thread:

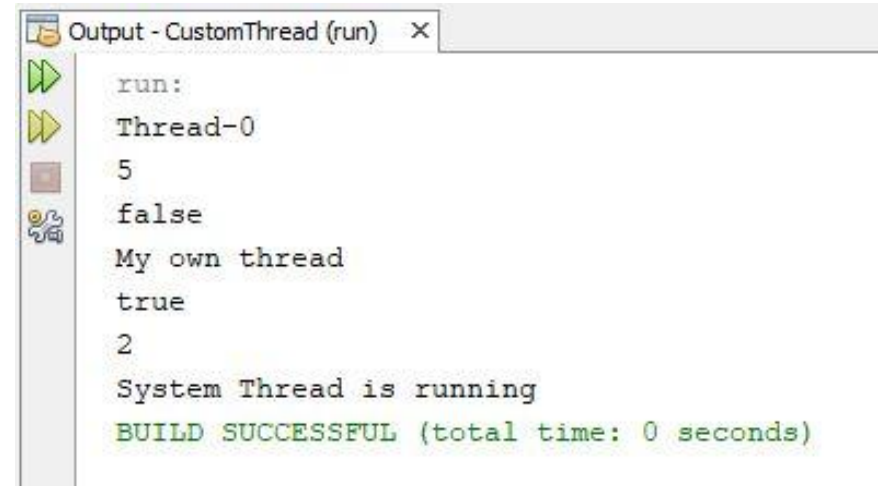
1. By extending the Thread class
2. By implementing Runnable Interface

❖ Java Thread Example by extending Thread class

1. Step 1: Override run() method available in Thread class. This method provides an entry point for the thread and we will put our complete business logic inside this method.
2. Step 2: Once Thread object is created, we can start it by calling start() method, which executes a call to run() method.

Creating a Thread-Example

```
public class MainClass {  
    public static void main(String[] args) {  
        CustomThread ct = new CustomThread();  
        System.out.println(ct.getName());  
        System.out.println(ct.getPriority());  
        System.out.println(ct.isAlive());  
        ct.setName("My own thread");  
        ct.start();  
        System.out.println(ct.getName());  
        System.out.println(ct.isAlive());  
        System.out.println(Thread.activeCount());  
    }  
}  
  
public class CustomThread extends Thread{  
    @Override  
    public void run(){  
        System.out.println("System Thread is running");  
    }  
}
```



```
Output - CustomThread (run) x  
run:  
Thread-0  
5  
false  
My own thread  
true  
2  
System Thread is running  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Creating a Thread-Method 2

❖ Create a Thread by Implementing a Runnable Interface.

1. Step 1

- As a first step, we need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and we will put out complete business logic inside this method.

2. Step 2

- As a second step, we will instantiate a Thread object using the following constructor.

`Thread(Runnable threadObj, String threadName);`

- Where the threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

3. Step 3

- Once a Thread object is created, we can start it by calling start() method, which executes a call to run() method.

Creating a Thread-Method 2

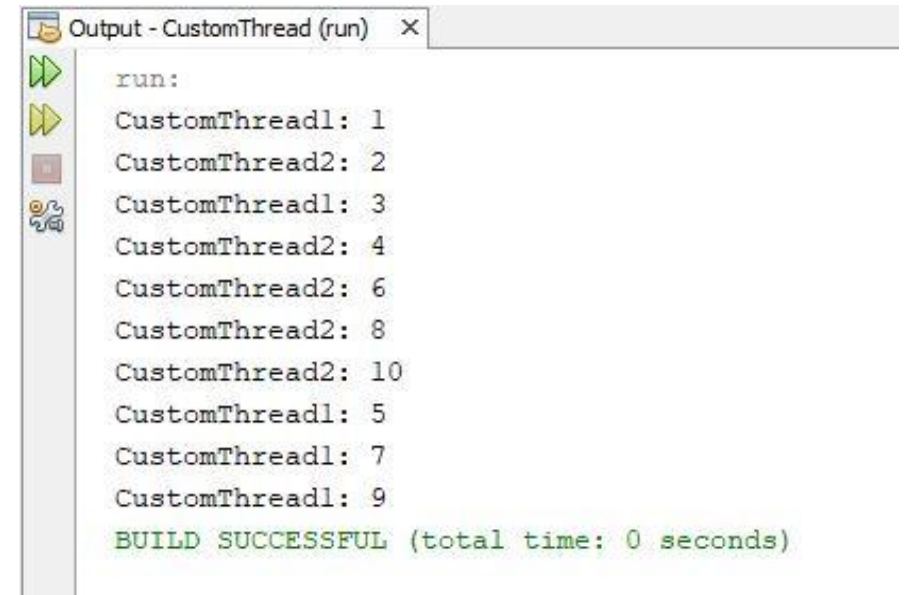
```
public class CustomThread1 implements Runnable{

    @Override
    public void run() {
        for(int i =1;i<10;i=i+2){
            System.out.println("CustomThread1: "+ i);
        }
    }
}
```

```
public class MainClass {
    public static void main(String[] args) {
        CustomThread1 ct1 = new CustomThread1();
        CustomThread2 ct2 = new CustomThread2();
        Thread t1 = new Thread(ct1);
        Thread t2 = new Thread(ct2);
        t1.start();
        t2.start();
    }
}
```

```
public class CustomThread2 implements Runnable {

    @Override
    public void run() {
        for(int i =2;i<=10;i=i+2){
            System.out.println("CustomThread2: "+ i);
        }
    }
}
```



Output - CustomThread (run) X

run:

CustomThread1: 1
CustomThread2: 2
CustomThread1: 3
CustomThread2: 4
CustomThread2: 6
CustomThread2: 8
CustomThread2: 10
CustomThread1: 5
CustomThread1: 7
CustomThread1: 9
BUILD SUCCESSFUL (total time: 0 seconds)

Thread Class vs Runnable Interface

- ❖ If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple Inheritance. But if we implement the Runnable interface, our class can still extend other base classes.
- ❖ We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like `yield()`, `interrupt()`, etc that are not available in Runnable Interface.
- ❖ Using Runnable will give us an object that can be shared amongst multiple threads.

Thread Priority

- ❖ Every Java Thread has a priority that helps the Operating System determine the order in which the threads are scheduled.
- ❖ Java Thread priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).
- ❖ Threads with higher priority are more important to a program and should be allocated processor time before lower priority threads.
- ❖ However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Thread Priority

❖ Check Thread Priority

```
System.out.println(Thread.currentThread().getPriority());
```

❖ Set Priority

```
Thread.currentThread().setPriority(9);
```

Threads Synchronization

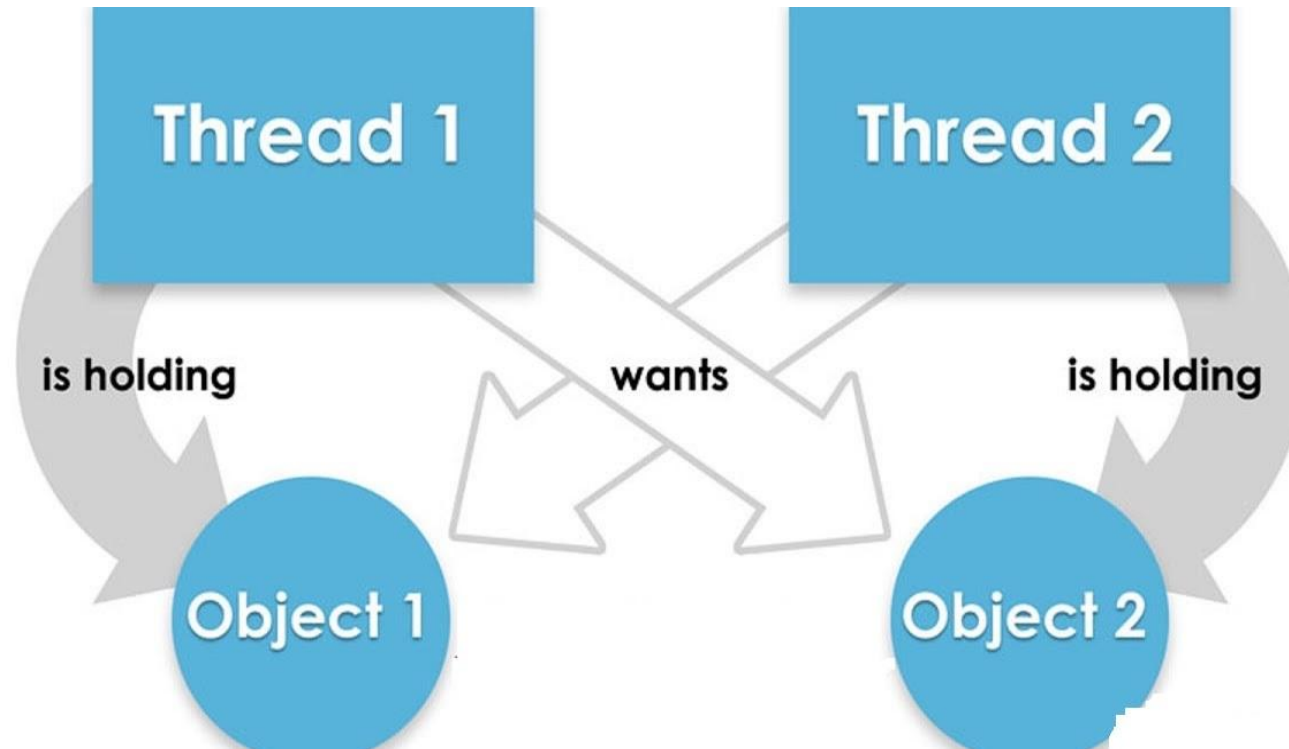
- ❖ When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.
- ❖ For example, when multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- ❖ So there is a need to synchronize the action of multiple threads and make sure that only one resource can access the resource at a given point in time.
- ❖ Thread Synchronization can be achieved by two types of approaches:
 1. Mutual Exclusion: It is a property of process synchronization which states that “no two processes can exist in the critical section at any given point of time”
 2. Inter-Thread Communication

Inter-Thread Communication

- ❖ Interthread communication is important when we develop an application where two or more threads execute some information.
- ❖ Threads may be paused running in its critical section and another thread is allowed to enter in the same critical section to be executed.
- ❖ There are three simple methods and a little trick which makes thread communication possible:
 - wait()
 - notify()
 - notifyAll()
- ❖ These methods have been implemented as final methods in Object class, so they are available in all the classes.
- ❖ All three methods can be called only from within the synchronized context.

Deadlock

- ❖ Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- ❖ Deadlock is a situation that occurs in OS when any thread enters a waiting state because another waiting thread is holding the demanded resource.



Deadlock

- ❖ Deadlock occurs when multiple threads need the same locks but obtain them in different order.
- ❖ A Java multithreaded program may suffer from deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with specified object.