Every **thread in Java is created and controlled** by a unique object of the java, lang. **Thread class**.
When a standalone application is **run**, a user thread is automatically created to execute the main() method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- By **implementing** the java.lang, **Runnable interface**
- By **extending** the java.lang. **Thread class**

To create and use a thread using this interface, we do the following:

1. Create a class which implements the Runnable interface. An object of this class is a Runnable object,
2. CreateanobjectoftypeThreadbypassingaRunnableobjectasargumenttotheThreadconstructor. The Thread object now has a Runnable object that implements the run( ) method,
3. The start() method is invoked on the Thread object created in the previous step.

Extending the Thread Class vs. Implementing the Runnable Interface
When creating threads, there are two reasons why implementing the Runnable interface may be prefer- able to extending the Thread class:

- Java does not support multiple inheritance. Therefore, extending the Thread class means that the subclass cannot extend any other class. A class implementing the Runnable interface will be able to extend another class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, which can be valuable. However, it also creates the opportunity for issues when two threads modify a resource at the same time. Java provides synchronization in order to control access to shared resources. The keyword synchronized and the lock form the basis for implementing synchronized execution of code.

Synchronized Methods
Most commonly, we restrict access to shared resources through the use of the s y n c h r o n i z e d keyword.It can be applied to methods and code blocks, and restricts multiple threads from executing the code simul- taneously on the same object.

Locks
For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds (or an equivalent situation with several threads). Since each

thread is waiting for the other thread to relinquish a lock, they both remain waiting forever. The threads are said to be deadlocked.

In order for a deadlock to occur, you must have all four of the following conditions met:

T. Mutual Exclusion: Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource, A deadlock could also occur if a resource has limited quantity.)
2. Hold and Wait: Processes already holding a resource can request additional resources, without relin- quishing their current resources.
3. No Preemption:One process cannot forcibly remove another process'resource.
4. Circular lVa/f;Two or more processes form a circular chain where each process is waiting on another resource in the chain.
Deadlock prevention entails removing any of the above conditions, but it gets tricky because many of these conditions are difficult to satisfy. For instance, removing #1 is difficult because many resources can only be used by one process at a time (e.g., printers). Most deadlock prevention algorithms focus on avoiding condition #4; circular wait.