Chapter 3

Multi threaded Programming

1. Introduction

- 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.
  - ✔ Each thread defines a separate path of execution.
- Multithreading is a specialized form of multitasking.
- There are two types of multitasking: process-based and thread-based.
- Process-based multitasking is the feature that allows your computer to run two or more programs concurrently.

1. Introduction

- Two tasks are operating concurrently means they’re both making progress at once.
- Two tasks are operating in parallel means they’re executing simultaneously.

- Processes are heavyweight tasks that require their own separate address spaces.
- In process-based multitasking,
  - Interprocess communication is expensive and limited.
  - Context switching from one process to another is costly
  - Process-based multitasking is not under Java’s direct control

1. Introduction

- In a thread-based multitasking a single program can perform two or more tasks simultaneously.
- Multitasking threads require less overhead than multitasking processes.
- In a thread-based multitasking environment,
  - ✔ Threads are lighter weight.
  - ✔ Threads share the same address space and cooperatively share the same heavyweight process.
  - ✔ Inter-thread communication is inexpensive.
  - ✔ Context switching from one thread to the next is lower in cost.
  - ✔ It is under java’s direct control.

2. Creating Threads

- All processes have at least one thread of execution, which is usually called the main Thread
- From the main thread, you can create other threads.
- You create a thread by instantiating an object of type Thread.
- The Thread class encapsulates an object that is runnable.
- Java defines two ways in which you can create a runnable object:
  - ✔ You can implement the Runnable interface.
  - ✔ You can extend the Thread class.
- To create a new thread, either extend Thread class or implement the Runnable interface.

2. Creating Threads

The Thread class

- The Thread class defines several methods that help to manage Threads. Some of its methods are:
  - final String getName()
  - final int getPriority()
  - final boolean isAlive()
  - void run() :- Entry point for the thread
  - void start() :- starts a thread by calling its run() method
  - static void sleep(long milliseconds):- suspends the thread
  - final void join() :- waits a thread to terminate.

2. Creating Threads

```java
class MyThread extends Thread{
    public MyThread(String name){ super(name); }
    public void run(){
        System.out.println(getName() + " starting");
        try{
            for(int i = 0; i<5 ; i++){
                Thread.sleep(1000);
                System.out.println(getName() + ", count: "  + i);
            } } catch(InterruptedException e){}
  }}
```

2. Creating Threads

```java
public class create {
    public static void main(String args[]) {
        System.out.println(Thread.currentThread.getName() + " start");
        MyThread thread1 = new MyThread("Thread 1");
        thread1.start();
        MyThread thread2 = new MyThread("Thread 2");
        thread2.start();
        System.out.println(Thread.currentThread.getName() + " end");
    }
}
```

2. Creating Threads

The Runnable interface

- You can construct a thread on any object that implements the Runnable interface.
- Runnable defines only one method called run( ):

```java
public void run()
```

- Inside run( ), you will define the code that constitutes the new thread.
- run( ) establishes the entry point for another concurrent thread of execution within your program.
- This thread will end when run( ) returns.

2. Creating Threads

```
class MyThread2 implements Runnable{
    public void run(){
        System.out.println(Thread.currentThread().getName() + " starting");
        try{
            for(int i = 0; i<5 ; i++){
              System.out.println(Thread.currentThread().getName() + ", count: " +i);
                Thread.sleep(1000);
            }
        }catch(InterruptedException e){}
   } }
```

2. Creating Threads

```
public class creat1 {
   public static void main(String args[]) {
       System.out.println(Thread.currentThread().getName() + " start" );
       MyThread2 thrd1 = new MyThread2();
       Thread mt1 = new Thread(thrd1);
       mt1.start();
       MyThread2 thrd2 = new MyThread2();
       Thread mt2 = new Thread(thrd2, "Adama");
       mt2.start();
        System.out.println(Thread.currentThread().getName() +" end");
   } }
```

2. Creating Threads

- We can add a Thread member variable and a constructor to MyThread class as:

```
Thread thrd; //a member variable of type Thread
 public MyThread(String name){ //a constructor
       thrd = new Thread(this, name);
 }
```

2. Creating Threads

- Then it’s run() method will be:

```
public void run(){
        System.out.println(thrd.getName() + " starting");
        try{
            for(int i = 0; i<5 ; i++){
                System.out.println(thrd.getName() + ", count: " +i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){}
    }
```

2. Creating Threads

- And the main() method will be:

```java
public static void main(String args[]) {
        System.out.println(Thread.currentThread().getName() + " start" );
        MyThread thread1 = new MyThread("Adama");
        thread1.thrd.start();
        MyThread thread2 = new MyThread("Hawassa");
        thread2.thrd.start();
        System.out.println(Thread.currentThread().getName() +" end");
    }
```

2. Creating Threads

- We can also add a factory method to MyThread class as:

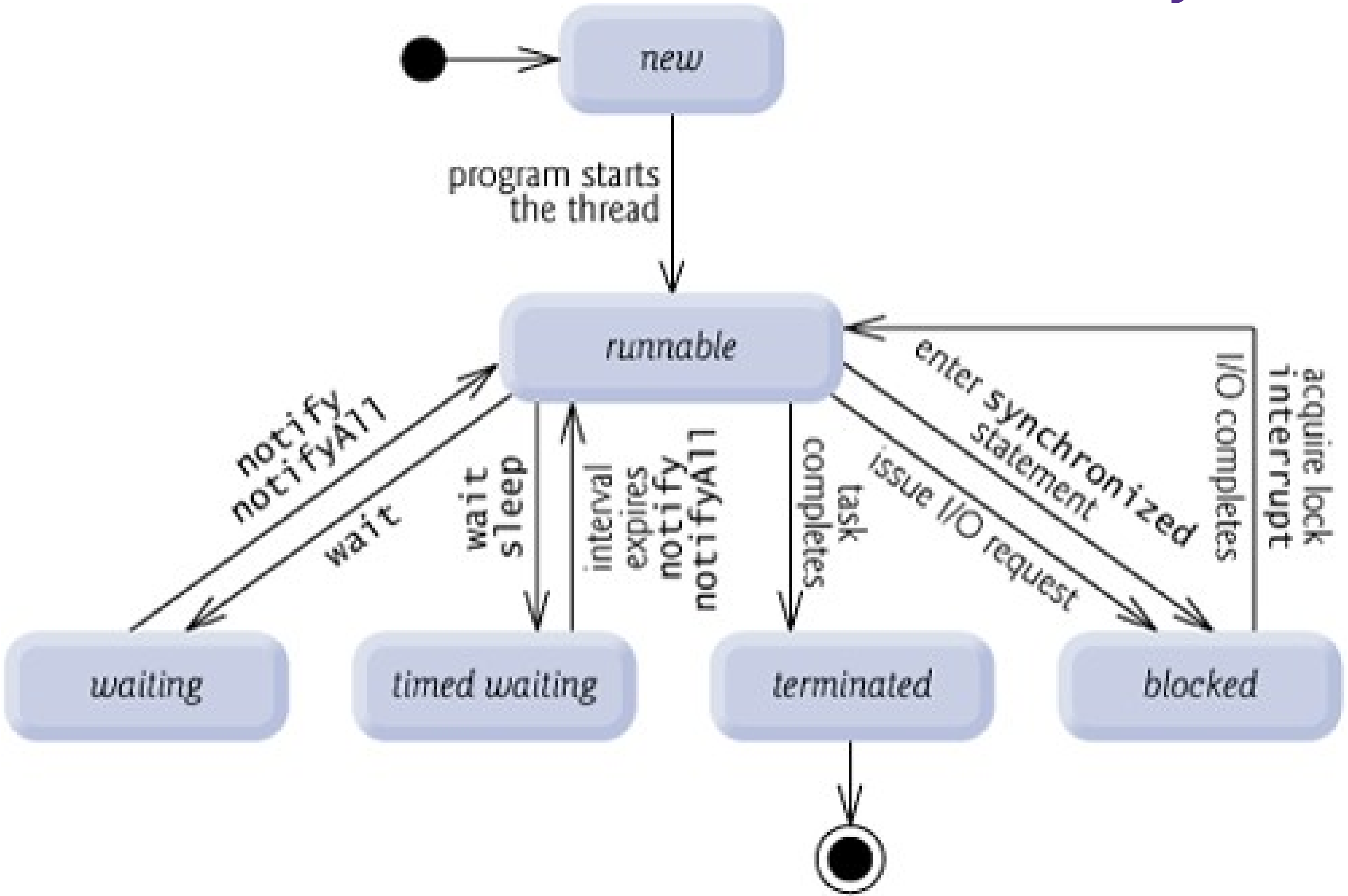
```
public static MyThread createThrd(String name){
        MyThread myThrd = new MyThread(name);
        myThrd.thrd.start();
        return myThrd;
    }
```

2. Creating Threads

- So that the main() method will be:

```java
public static void main(String args[]) {
        System.out.println(Thread.currentThread().getName() + " start" );
        MyThread.createThrd("Adama");
        MyThread.createThrd("Hawassa");
        System.out.println(Thread.currentThread().getName() +" end");
}
```

3. Thread States and Life Cycle



3. Thread States and Life Cycle

- At any time, a thread is said to be in one of several thread states

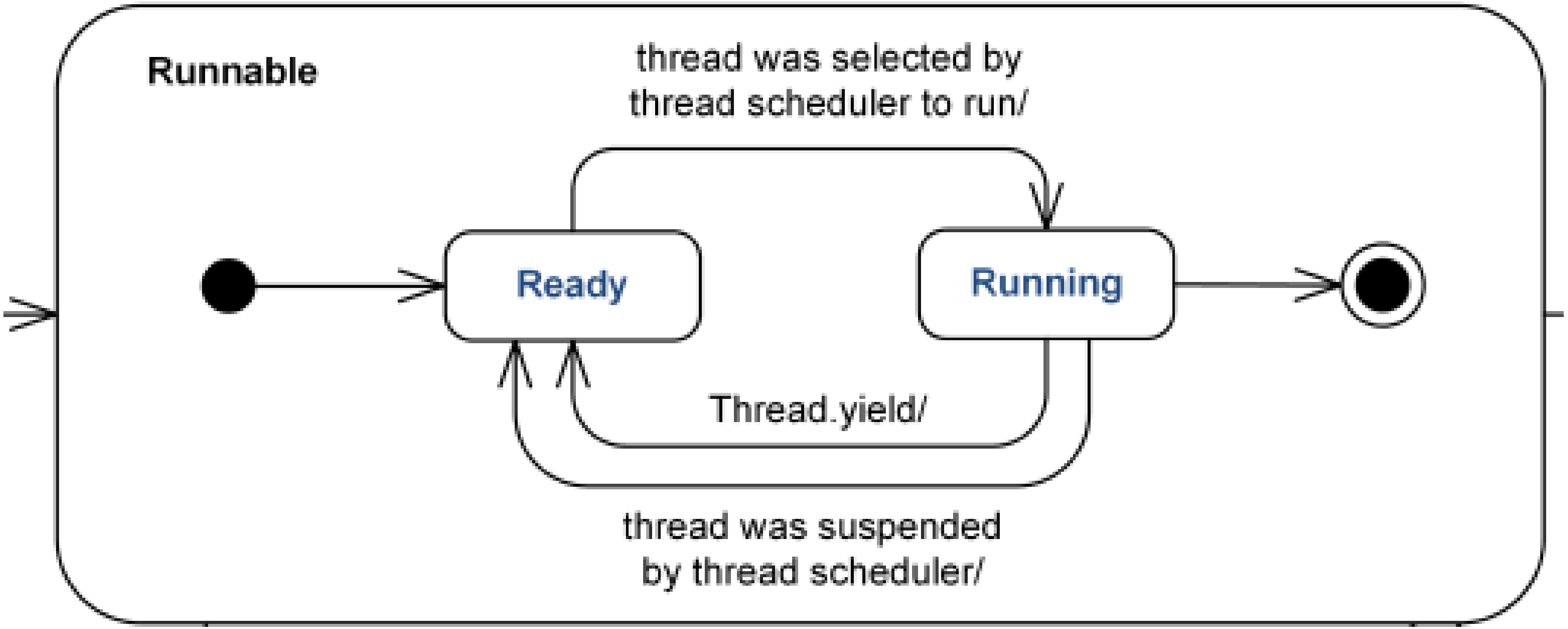
New State:

- When a thread has just been created, it is in the "New" state.
  - the Thread is not yet scheduled for execution and has not started running.

"Runnable" State:

- A thread enters the "Runnable" state after invoking the start() method.
- A thread in the runnable state is either
  - Executing its task or.
  - Is ready to run and waiting to be selected by thread scheduler to run.

3. Thread States and Life Cycle



3. Thread States and Life Cycle

Waiting State

- A runnable thread can transition to the waiting state while it waits for another thread to perform a task.
- A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.

Timed Waiting State

- A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it's waiting for another thread to perform a task.
- It returns to the runnable state when it's notified by another thread or when the timed interval expires.

3. Thread States and Life Cycle

- Another way to place a thread in the timed waiting state is to put a runnable thread to sleep
  - ✔ Returns to the runnable state when the sleep interval expires.

Blocked state

- A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately.
- For example, when a thread issues an input/output request.
  - ✔ OS blocks the thread until that I/O request completes

Terminated State

- A runnable thread enters the terminated or dead state when it successfully completes its task or terminates due to an error.

4. Creating Threads with The Executor

- Runnable specify a task that execute concurrently with other tasks.
- An Executor object executes Runnables by Creating and managing a group of threads called a thread pool.

Executor interface

- has method execute, which accepts a Runnable as an argument.
- Assigns every Runnable passed to its execute method to one of the available threads in the thread pool.
  - ✔ If there are no available threads, it creates a new thread or waits for a thread to become available.
- To execute a Runnable, it calls its run method.

4. Creating Threads with The Executor

The ExecutorService interface

- extends Executer interface and declares various methods for managing the life cycle of an Executor.
- Its object can be obtained by calling one of the static methods declared in class Executors
  - ✔ Executors class provides several static methods that can be used to create and manage a thread pool.
- Advantages of using Executor over creating threads yourself.
  - ✔ Executors can reuse existing threads to eliminate the overhead of creating a new thread for each task
  - ✔ Executor improve performance by optimizing the number of threads

4. Creating Threads with The Executor

```
class MyThread3 implements Runnable{
    String thrdName;
    public MyThread3(String name){
        thrdName = name;
    }
    public void run(){
        System.out.println(thrdName + " starting");
        try{
            for(int i = 0; i<5 ; i++){
                System.out.println(thrdName + ", count: " +i);
                Thread.sleep(1000);
            }
        }catch(InterruptedException e){}
    } }
```

4. Creating Threads with The Executor

```
public static void main(String args[]) {
        System.out.println(" Main start");
        ExecutorService es = Executors.newCachedThreadPool();
        MyThread3 task1 = new MyThread3("Task #1");
        MyThread3 task2 = new MyThread3("Task #2");
        MyThread3 task3 = new MyThread3("Task #3");
        es.execute(task1);
        es.execute(task2);
        es.execute(task3);
         System.out.println(" Main start");
}
```

5. Thread Properties

Thread Priority

- Every Java thread has a thread priority that helps determine the order in which threads are scheduled.
- Each new thread inherits the priority of the thread that created it.
- You can set the priority of a thread by using setPriority(int n) method of Thread class.
- Higher-priority threads gets processor time before lower-priority threads.
  - ✔ Nevertheless, thread priorities cannot guarantee the order in which threads execute.

5. Thread Properties

Starvation

- Is the indefinite postponement of the execution of threads by higher-priority threads.
- To prevent starvation, operating systems gradually increase the lower priority thread’s priority so that it will eventually run.

Deadlock

- Occurs when to threads wait for each other simultaneously (directly or indirectly) to proceed.

5. Thread Properties

Daemon Threads

- Is a thread that has no other role in life than to serve others.
- Runs in the background and does not prevent the JVM from exiting when all non-daemon threads in Java have been completed.
- To turn a thread to a daemon thread, use setDaemon() method as: t.setDaemon(true)

Thread Name

- Threads have default names
- You can also set any name to a thread with Thread's setName method.

5. Thread Properties

Thread Join

- join() method can be used to pause the current thread execution until unless the specified thread is dead.
  - It puts the current thread on wait until the thread on which it’s called is dead.
- join(long millis): is used to wait for the thread on which it’s called to be dead or wait for specified milliseconds.

Yield() method

- causes the currently executing thread object to temporarily pause and allow other threads to execute.

5. Thread Properties

```java
public static void main(String args[]) {
    System.out.println("Main start" );
    Thread mt1 = new Thread(new MyThread2());
    mt1.start();
    Thread mt2 = new Thread(new MyThread2(), "Adama");
    mt2.start();
    try{
        mt1.join();//main waits until thread mt1 is dead
        mt2.join(5000);//main waits until mt2 is dead or up to 5 seconds
     }catch(InterruptedException e){}
     System.out.println("Main end");
 }
```

6. Synchronization

- Synchronization is the capability to control the access of multiple threads to any shared resource.
- Common reason for synchronization is when two or more threads need access to a shared resource that can be used by only one thread at a time.
  - For example, when two threads want write to a file at the same time.
- Another reason for synchronization is when one thread is waiting for an event that is caused by another thread.
  - For example producer consumer
- All objects in Java have a monitor and a monitor lock, which controls access to an object.

6. Synchronization

- The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time.
  - When an object is locked by one thread, no other thread can gain access to the object.
  - When the thread exits, the object is unlocked and is available for use by another thread.
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement.

```
synchronized (object) {
    statements
}
```

6. Synchronization

```
class MyArray{
    private int[] myArr;
    private int position = 0;
    private SecureRandom sr = new SecureRandom();
    public MyArray(int size){ myArr = new int[size]; }
    public void add(int value){
        int index = position;
        try{ Thread.sleep(sr.nextInt(500));}
        catch(InterruptedException e){}
         myArr[index] = value;
         System.out.println(Thread.currentThread().getName()+
                        " wrote "+ value + " at index " + index);
         position++;
```

6. Synchronization

```
class MyThread implements Runnable{
    MyArray myArr;
    int startValue;
    public MyThread(int value, MyArray arr){
        myArr = arr;
        startValue = value;
    }
    public void run(){
        for(int i = startValue; i < startValue+3; i++)
            myArr.add(i);
     }
}
```

6. Synchronization

```
public class sync {
    public static void main(String args[]) {
        MyArray myArr = new MyArray(6);
        Thread mt1 = new Thread(new MyThread4(1, myArr));
        Thread mt2 = new Thread(new MyThread4(17, myArr));
        mt1.start();
        mt2.start();
    }
}
```

6. Synchronization

- In the above program an object of MyArray class is shared among two threads
- MyArray enable these threads to put and int value to array.
- The threads have unsynchronized access to the array.
- i.e the two threads may access the array concurrently.
- A value which was written to the array by a thread may be overwritten later by the the other thread.
- There is unpredictability of thread scheduling and to increase the likelihood of producing erroneous output.

6. Synchronization

- The problem lies in method add, which stores the position value, places a new value in that index, then increments position.
- If one thread obtains the position value, another thread may come along and increment position before the first thread to place a value in the array.
- MyArray allows any number of threads to read and modify shared mutable data concurrently
- To ensure that no two threads can access its shared mutable data at the same time. Make the access to array atomic operation.
- Atomicity can be achieved using the synchronized keyword.
- Make add method synchronized.

6. Synchronization

```
class MyArray{
    private int[] myArr;
    private int position = 0;
    private SecureRandom sr = new SecureRandom();
    public MyArray(int size){ myArr = new int[size]; }
    public synchronized void add(int value){
        int index = position;
        try{ Thread.sleep(sr.nextInt(500));}
        catch(InterruptedException e){}
         myArr[index] = value;
         System.out.println(Thread.currentThread().getName()+
             " wrote " + value + " at index " + index);
         position++;

```

7. Thread Communication

- Java supports inter-thread communication with the wait( ), notify( ), and notifyAll( ) methods.
  - These methods are implemented by Object class – are part of all objects
  - These methods should be called only from within a synchronized context.
- When a thread calls wait( ).
  - The thread go to sleep and release the monitor for that object.
    - Allowing another thread to use the object.
  - At later point, the sleeping thread is awakened when some other thread enters the same monitor and calls notify( ), or notifyAll( ).

7. Thread Communication

- A call to notify( ) resumes one waiting thread.
- A call to notifyAll( ) notifies all threads.
  - the scheduler determining which thread gains access to the object.
- waiting thread could be awakened due to a spurious wakeup.
  - calls to wait( ) should take place within a loop that checks the condition on which the thread is waiting.

```
while(condition){
      try{
          wait();
      }catch(InterruptedException e){}
}
```

7. Thread Communication

```
class TickTock{
    String state = "Tick"; //initial state is Tick
    synchronized void tick(){  //prints Tick
        while(state.compareTo("Tock")== 0){ //wait while state is Tock
            try{
                wait();
            }catch(InterruptedException e){}
        }
        System.out.print("Tick "); //print Tick
        state = "Tock"; //set state to Tock
        Notify(); //notify the other thread
    }
```

7. Thread Communication

```
synchronized void tock(){  //prints Tock
        while(state.compareTo("Tick") == 0){ //wait while state is Tick
            try{
                wait();
            }catch(InterruptedException e){}
        }
        System.out.println("Tock "); //print Tock
        state = "Tick"; //set state to Tick
        Notify();  //notify the other thread
    }
}
```

7. Thread Communication

```
class ThreadTT implements Runnable{
    TickTock tt;
    ThreadTT(TickTock tt){
        this.tt = tt;
    }
    public void run(){
        if(Thread.currentThread().getName().compareTo("ticker") == 0)
            for(int i = 0; i < 5; i++)
                tt.tick();
        else if(Thread.currentThread().getName().compareTo("tocker")==0)
            for(int i = 0; i < 5; i++)
                tt.tock();
    }}
```

7. Thread Communication

```java
public class wait {
    public static void main(String args[]) {
        TickTock tt = new TickTock();
        ThreadTT tick = new ThreadTT(tt);
        ThreadTT tock = new ThreadTT(tt);
        Thread ticker = new Thread(tick, "ticker");
        Thread tocker = new Thread(tock, "tocker");
        ticker.start();
        tocker.start();
    }
}
```

8. Thread-safe Collections

- A thread-safe class is a class that always maintains valid state even when used concurrently by multiple threads.
- Thread-safe collections are designed to be thread-safe.
  - They can be safely used by multiple threads without the need for external synchronization.
- A thread-safe collection can be synchronized collection or a concurrent collection.
- Concurrent collections achieve thread-safety by partitioning the data into segments.
  - Threads can access these segments concurrently and obtain locks only on the segments that are used.

8. Thread-safe Collections

- Synchronized collection locks the entire collection via intrinsic locking
  - Has low performance than Concurrent Collection
    - At a time only one thread is allowed to operate on an object so it increases the waiting time of the threads.
- Java 1.0 legacy classes like HashTable, Vector, and Stack are synchronized collections
- Java 1.2 collections are unsynchronized.
  - The Collections class provides static methods for wrapping collections as synchronized versions.
  - These are wrappers that returns a thread-safe collection backed up by the specified Collection.

8. Thread-safe Collections

```
List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());

Map<Integer, String> syncMap = Collections.synchronizedMap(new
HashMap<>());

Set<Integer> syncSet = Collections.synchronizedSet(new HashSet<>());
```

- There are concurrent collections in java.util.concurrent package.
  - Allow multiple threads to access and modify a collection concurrently, without the need for explicit synchronization.
  - Provide thread-safe implementations of the traditional collection interfaces like List, Set, and Map.

8. Thread-safe Collections

- Examples of concurrent collection classes include
  - ArrayBlockingQueue
    - Implements BlockingQueue interface
    - A fixed-size queue that supports the producer/consumer relationship
    - Has methods put and take
    - put places an element at the end of the BlockingQueue, waiting if the queue is full.
    - take removes an element from the head of the BlockingQueue, waiting if the queue is empty.
  - ConcurrentHashMap
    - A hash-based map (similar to the HashMap)

8. Thread-safe Collections

```
class MyBuffer{
    ArrayBlockingQueue<Integer> queue; //is thread-safe, no need of
    MyBuffer(){                                    //synchronization
        queue = new ArrayBlockingQueue<>(1); //size 1
    }
    void myPut(int value) throws InterruptedException{
        queue.put(value); //put value to array blocking queue
        System.out.println(Thread.currentThread().getName()+" writes " +value);
    }
    Integer myTake() throws InterruptedException{
        int n = queue.take(); //get from array blocking queue
        System.out.println(Thread.currentThread().getName()+" reads " +n);
        return n;
    }}
```

8. Thread-safe Collections

```
class Producer implements Runnable{
    MyBuffer mb;
    String thrdName;
    Producer(MyBuffer mb, String name){
        this.mb =mb;
        thrdName = name;
    }
    public void run(){
        Thread.currentThread().setName(thrdName);
        for(int i=1; i<=5; i++)
            try{
                mb.myPut(i);
            }catch(InterruptedException e){}
    }}
```

8. Thread-safe Collections

```
class Consumer implements Runnable{
    MyBuffer mb;
    String thrdName;
    Consumer(MyBuffer mb, String name){
        this.mb =mb;
        thrdName = name;
    }
    public void run(){
        Thread.currentThread().setName(thrdName);
        for(int i=1; i<=5; i++)
            try{
                mb.myTake();
            }catch(InterruptedException e){}
    }
}
```

8. Thread-safe Collections

```
public class ConcurrentCollection {
    public static void main(String args[]) {
        MyBuffer mb = new MyBuffer();
        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(new Consumer(mb, "Consumer 1"));
        es.execute(new Producer(mb, "Producer 1"));
        es.execute(new Consumer(mb, "Consumer 2"));
        es.execute(new Producer(mb, "Producer 2"));
        es.shutdown();
    }
}
```