Threads

Objectives

- Appreciate Threads and use it appropriately in programs

Table of Content

What is a thread?

- Oracle/Sun defines a thread as a single sequential flow of control within a program.
- The thread in java is a realization of OS level thread. In other words, the thread in java is created using native methods which in turn create threads based on the OS.
- Threads are used in
  - Client-server based systems, the server program creates threads that allows it to respond to multiple users at the same time.
  - GUI programs have a separate thread to gather user's interface events from the host OS.
  - doing other things while waiting for slow I/O operations.
  - animations

`java.lang.Thread` class

- Java SE provides 2 classes for working with thread called **`Thread`** and **`ThreadGroup`**.
- The main threads continues to execute until
  - **`exit()`** method is called
  - all threads (non-daemon threads) have died
- A deamon thread is a special type of thread that runs in background. When JVM exits, only remaining thread which will be running are daemon threads. By default all the threads that are created in java are non-daemon threads.

Printing main thread

```
public class ThreadTest{
public static void main(String s[])
{
System.out.println("Hello");
System.out.println(
      Thread.currentThread().getName());

}
}
```

Prints : main

currentThread() is a _____ method of _____class.
getName() is _____ method of _____ class

`Thread` class constructors

- There are two ways to create threads
  - By extending **`Thread`** class
    - Constructors that will be called in such case could be
      - **`Thread()`**
      - **`Thread(String name)`**
  - By creating **`Thread object`** and passing **`Runnable`** instance.
    - Constructors that will be called in such case could be
      - **`Thread(Runnable target)`**
      - **`Thread(Runnable target, String name)`**

Creating Threads by extending

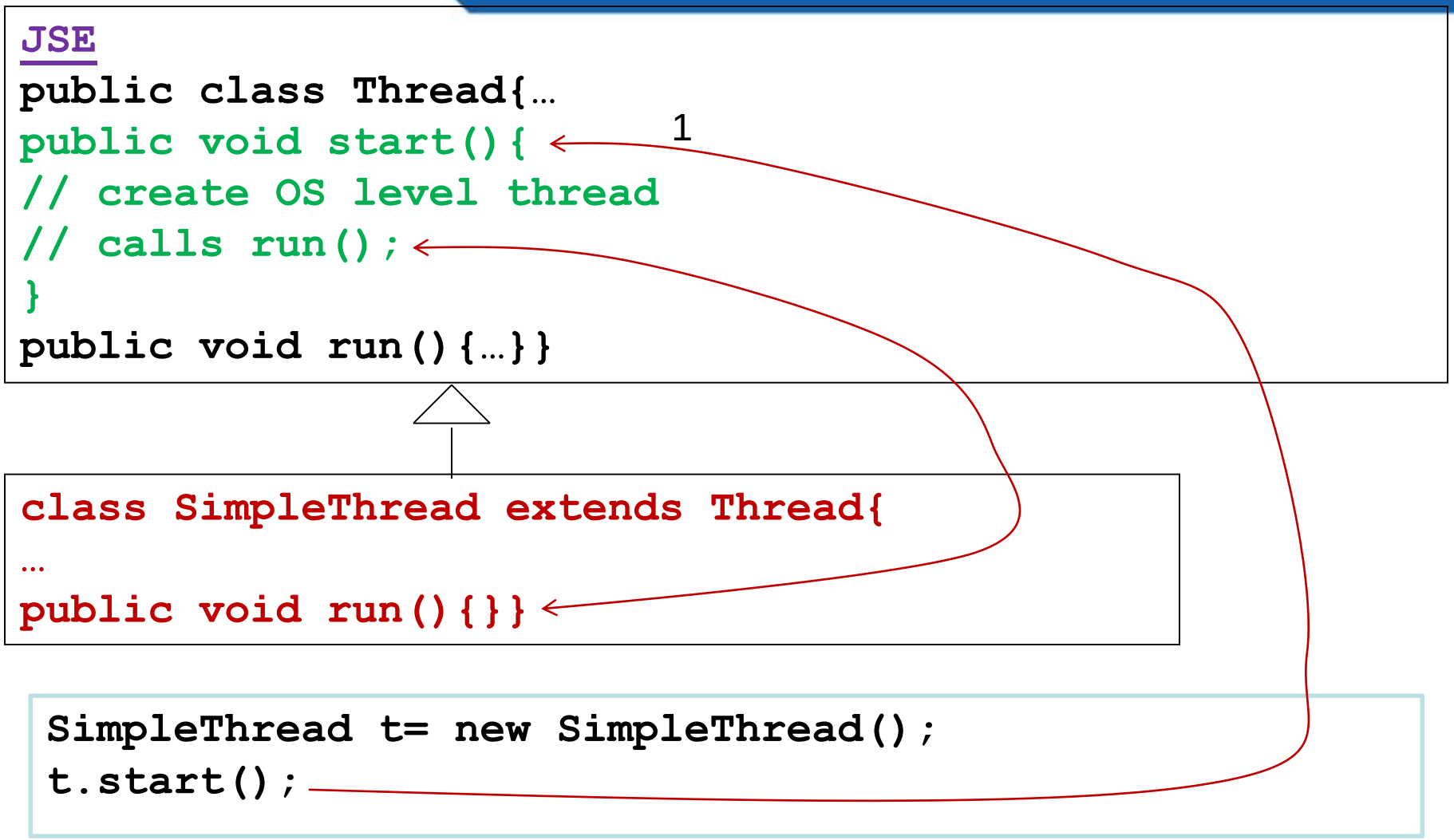
```
class SimpleThread extends Thread {
public void run(){
/* code that is executed when
thread executes */
}}
```

Creating the **`SimpleThread`** object

```
SimpleThread t= new SimpleThread();
t.start(); // Calls run() method of SimpleThread
```

- The **`start()`** method of the **`Thread`** class creates OS level thread.
- After this it calls the **`run()`** method.

Starting a thread



Create Thread through inheritance

```
import java.util.*;
class PrimeOddThread extends Thread {
int num;
public static void main(String str[]){
      PrimeOddThread c= new PrimeOddThread ();
      Scanner scan= new Scanner(System.in);
      int i=scan.nextInt();
      c.num=i;
      c.start();
      if(c.odd())
     System.out.println("Odd");
      else  System.out.println("Even");
}
```

```
public void run() {
      if( prime())
   System.out.println("Prime");
   else      System.out.println("Non-Prime");
  }
public boolean prime(){
      for(int i=2;i<num/2;i++){
      if(num%i==0)              return false;}
      return true;}
public boolean odd(){
      if(num%2==0) return false;
      else return true;
  }}
```

It is totally up to the OS thread scheduler to choose which thread to execute first.

What will happen if you call ***`run()`*** *method instead of* ***`start()`*** *method in the previous example?*

Exercise

- *The main method waits to get input from the users until it is terminated. The input that it receives is any string. As soon as the main method receives the input it delegates the assignment of creating a password to a thread. The thread generates a random number and appends this number to the string that is passed and displays the password. Write a java program to do this.*
- *Hint : Use java.util.Random class to generate random numbers.*

(45 mins)

Using `Runnable`

- **`Runnable`** interface has one method
  - **`public void run()`**
- Second way to create thread is by
  1. Creating a class that implements **`Runnable`** interface- that is overriding the **`run()`**
  2. Creating a Thread class instance and passing **`Runnable`** instance.

```
class SimpleThreadR implements Runnable{
public void run(){…}
}
```

*//*Creation of thread – using a constructor that expects a Runnable object

```
Thread t= new Thread(new SimpleThreadR() );
t.start(); // calls the run method of SimpleThreadR
```

Callback working behind the scene

```
class Thread implements Runnable{…
private Runnable target;
public Thread(Runnable target){
        this.target=target;
…
}
public void start(){
        // create OS level thread
        run();
}
public void run(){
if (target != null){target.run();    }}
```

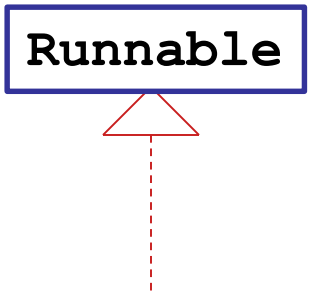
1

2

3

```
Thread t=new Thread(new
SimpleThreadR() );
t.start();}
```

```
class SimpleThreadR
implements Runnable{…
public void run(){}}
```



Activity

- Locate src zipped file in Java installation directory. Extract the file, locate Thread.java. Open the file. Go through and understand the run() and start() methods.

Example :create Thread using Runnable

```
import java.util.*;
class PrimeOddThread implements Runnable {
int num;

public static void main(String str[]){
    PrimeOddThread x=new PrimeOddThread();
    Thread c= new Thread (x );
    Scanner scan= new Scanner(System.in);
    int i=scan.nextInt();
    x.num=i;
    c.start();
    if(x.odd())
          System.out.println("Odd");
    else
    System.out.println("Even");
}
```

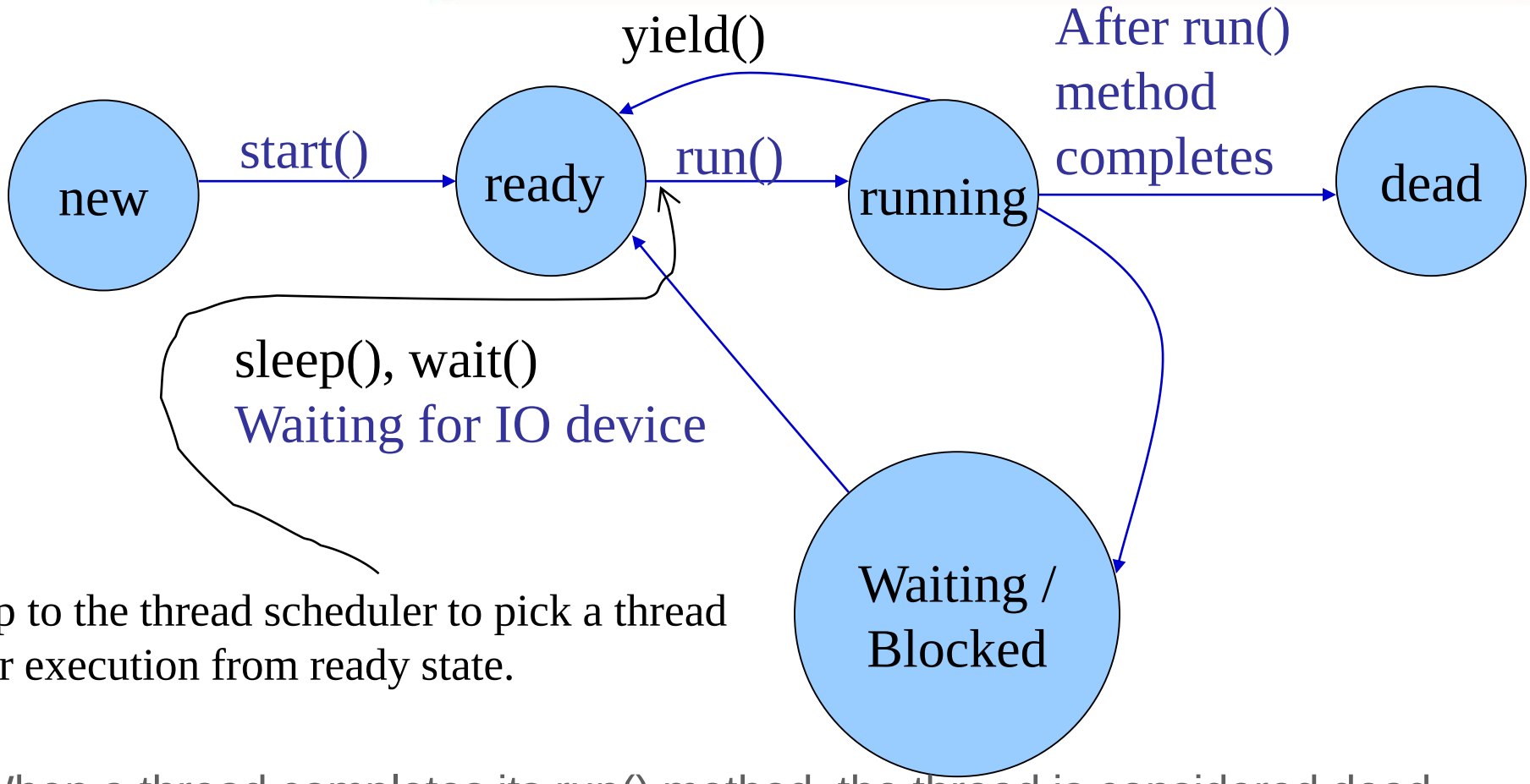
```
public void run() {
    if( prime())
      System.out.println("Prime");
      else System.out.println("Non-Prime");
}

public boolean prime(){
   for(int i=2;i<num/2;i++){
   if(num%i==0)return false;}
   return true;}
   public boolean odd(){
   if(num%2==0) return false;
   else return true;
}
}
```

Exercise

- *Implement the previous exercise using Runnable.*

(15 mins)

Lifecycle



- When a thread completes its run() method, the thread is considered dead.
- **`final boolean isAlive()`** can be used to check if the thread is alive or dead.
- An attempt to start a dead thread throws **`IllegalThreadStateException`**

Putting a thread to sleep

- **`sleep()`** causes the currently executing thread to sleep (temporarily cease execution) for the minimum of specified number of milliseconds.
- **`static void sleep(long millis) throws InterruptedException`**
- **`static void sleep(long millis, int nanos) throws InterruptedException`**
- Note that **`sleep()`** is **`static`** method.
- **`InterruptedException`** is a checked exception. A sleeping thread can be interrupted so that it's sleep can be broken. This can be done using **`interrupt()`**

Activity

```
public class Appear implements Runnable{
 char c[]={'H','E','L','L','O'};
 public void run() {
      int i=0;
      while (i<5){
      System.out.print(c[i++]);
      }
  }
 public static void main(String str[]){
      Thread t =new Thread(new Appear());
      t.start();
 }}
```

What can be added to the above code so that the characters are printed after every 1 second? Compile and verify that it works.

Interruption

- A thread can be interrupted while it is sleeping or waiting. This can be done by calling **`interrupt()`** on it. In other words, when the thread is calling **`sleep(), (join() or wait() )`**, it can be interrupted .
- When **`interrupt()`** is called, an **`InterruptedException`** exception will be thrown. So the catch handler block will be executed. **`sleep(), join() or wait()`** methods throw **`InterruptedException`**
- The **`interrupt`** status will be reset before the **`InterruptedException`** exception is thrown.
- Methods that can be used to test if the current thread has been interrupted
  - **`static boolean interrupted() : interrupted status`** of the thread is cleared by this method
  - **`boolean isInterrupted(): interrupted status`** of the thread is unaffected

Exercise

- *A prompt asking a question appears for which user is given 1 minute. If user answers the question before 1 minute then "Congratulations!" is displayed. Otherwise "Better Luck Next Time" is displayed.*
- *Write a program to implement the above scenario.*

(30 mins)

Example: `join()`

If we want to guarantee that Prime (or Non-Prime) is printed before Even(or Odd) then the thread printing Even(or Odd), that is main thread, can wait till the prime thread finishes the execution.

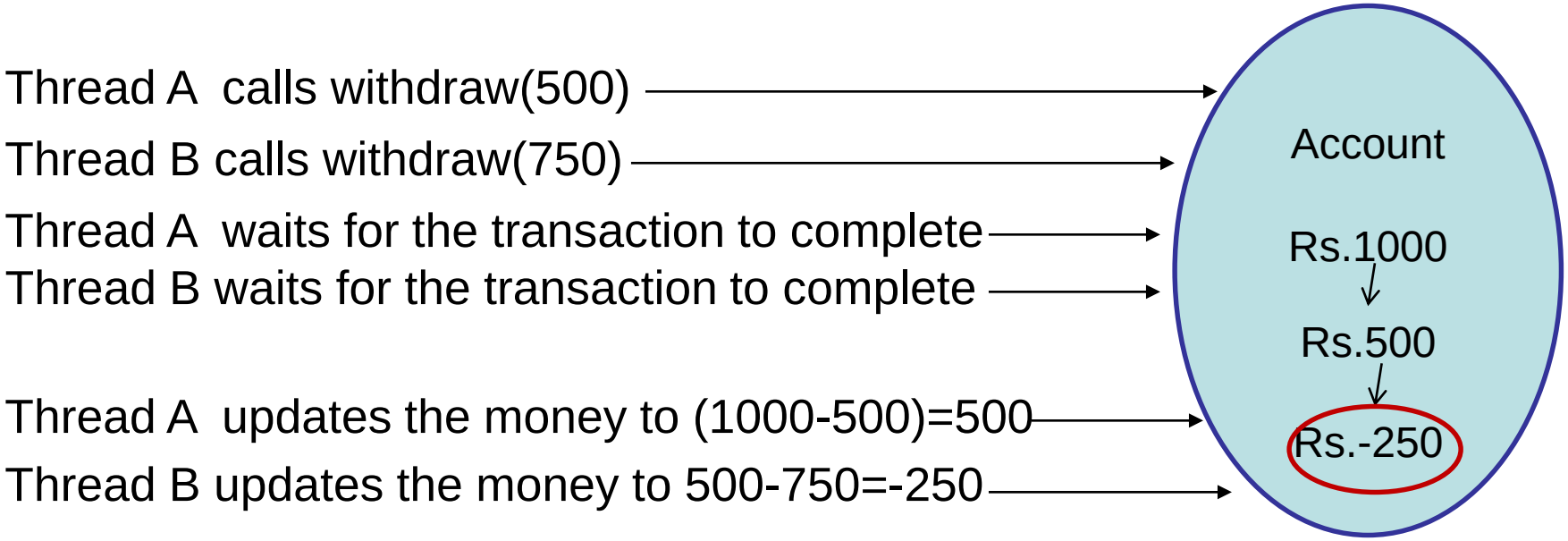
```
class Join implements Runnable{
int num;
public void run() {
 if( prime()) System.out.print("Prime");
 else         System.out.print("Non-Prime"); }

public boolean prime(){
   for(int i=2;i<num/2;i++){
   if(num%i==0)return false;
   }
   return true;   }

public boolean odd(){
   if(num%2==0)return false;
   else return true;    }
```

```
public static void main(String str[]){
    Join s=new Join();
    Thread t= new Thread(s);
    s.num=55;
    t.start();

    try{ t.join();} //main thread waits for t to finish

    catch(InterruptedException  e){}


    boolean b=s.odd();

    if(b)

    System.out.print(" and Odd");
    else     System.out.print("and Even");

}}
```

Thread priorities

- A thread is always associated with a priority based on the OS
- For simplicity Java assigns priority as a number between 1 and 10, inclusive of 10. 10 is the highest priority and 1 is the lowest.
- This is the only way to influence the scheduler's decision as to the thread execution sequence to some extent.
- At any given time, the highest-priority thread will be chosen to run. However, this is not guaranteed. The thread scheduler may choose to run a lower-priority thread to avoid starvation.
- Threads that were created so far were created with a default priority of 5.

Without synchronization?



Code demonstrating the same…

Code without synchronization

```
class Account{
   private int money;
   Account(int amt){
   //get amt from database
      money=amt;}
   void withdraw(int amt){
      if(amt<money){
      try{
         /*
          sleep() here is used to simulating time to
         connect to other systems and performing IO
         operation
         */
      Thread.sleep(1000);
       money=money-amt;
       }catch(Exception e){}
```

```
    System.out.println("Received "+ amt  +" by " +
    Thread.currentThread().getName());
        }
    else
    System.out.println("Sorry "+
    Thread.currentThread().getName()+ "Requested amt ("+
    amt +") is not available.");
    System.out.println("Balance "+ money);
}}
public class ThreadTest implements Runnable{
    Account a;
    int amt;
public static void main(String str[]){
  Account lb= new Account(1000);
  new ThreadTest(lb,"A",500);
  new ThreadTest(lb,"B",750);
}
```

```
public ThreadTest(Account a,String name,int amt){
  this.a=a;
  this.amt=amt;
  new Thread(this,name).start();
}

public void run(){ a.withdraw(amt);}
}
```

```
Result:
Received 500 by A
Balance 500
Received 750 by B
Balance -250
```

Terms

- If two threads access the same object and each calls a method that changes the state of that object then data corruption could result. This is called **race condition.**
- Monitor is a block of code guarded by a mutual-exclusion semaphore (*mutex or intrinsic lock* or *monitor lock*). A thread that wants to enter the monitor must have mutex.
- Only one thread can own the mutex at a time. Other threads which wants to enter the critical code must wait till the thread that acquired mutex releases it.
- In Java, every *object* has one and only one monitor and mutex associated with it.

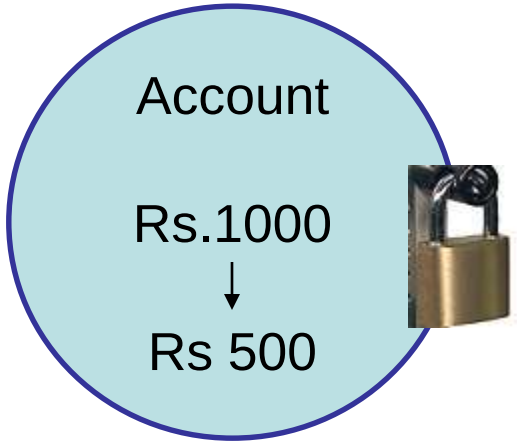
Solution to the Account problem

Thread A acquires mutex for the Account object and calls withdraw(500)

Thread B waits for A to release the mutex

Thread A waits for the transaction to complete

Thread A updates the money to (1000-500)=500 and releases the mutex



Thread B acquires mutex for the object and calls withdraw(750)

Thread B gets the message that it cannot withdraw the requested amount

Thread B releases the mutex

synchronized

- Java implements this concept by using **synchronized** keyword.
- **synchronized** can be used in 2 ways
  - A method can be marked **synchronized**
  - A block of code can be marked **synchronized**→ **synchronized** statements
- A thread enters a **synchronized** block of code only if no other thread is executing any other a **synchronized** block of code on that object. Therefore, two invocations of synchronized methods on the same object cannot interleave.
- Once a thread enters a **synchronized** code, it acquires a mutex. It is released only when the thread leaves the **synchronized** code.

Correcting code using `synchronized`

- If an object is visible to more than one thread, all reads or writes to that object's non final attributes should be done through synchronized methods.

- Approach 1: Add synchronized keyword to withdraw and other critical methods of the Account object.

```
synchronized void withdraw(int amt)
```

- Approach 2: Use `synchronized` statements by explicitly locking the object before calling critical methods of the Account object.

```
public void run(){
  synchronized(a)     {
   a.withdraw(amt);   }
  }
```

Exercise

- *Write an application to simulate the vehicles crossing a bridge and a toll plaza on a highway. For the purpose of this exercise, simulate the environment for five vehicles that are approaching the bridge and the toll booth. The vehicles are numbered from one to five. The vehicles should approach the bridge and the toll booth in sequential order. The toll booth can only deal with one vehicle at a time. This application should print a message every time when a vehicle crosses the bridge and another message when a vehicle crosses the toll booth along with the vehicle number.*

(45 mins)

Code leading to deadlock

```
public class Lock{
  public static void main(String[] args) {
    final Account resource1 = new Account(2000);
    final Account resource2 = = new Account(2000);

 // t1 tries to lock resource1 then resource2
    Thread t1
= new Thread() {

 public void run() {
        // Lock resource 1
        synchronized (resource1) {
      System.out.println("Thread 1:locked resource 1.
updating the balance");
      try {
            Thread.sleep(50);
          } catch (InterruptedException e) {   }
```

```
// Lock resource 2
synchronized (resource2) {
System.out.println("Thread 1: locked resource 2.
updating the balance "); }
 }
 } };

// t2 tries to lock resource2 then resource1
    Thread t2 = new Thread() {
      public void run() {
      // Lock resource 2
        synchronized (resource2) {
          System.out.println("Thread 2: locked
resource. updating the balance ");
try {
            Thread.sleep(50);
          } catch (InterruptedException e) {
          }
```

```
// Lock resource 1
synchronized (resource1) {
            System.out.println("Thread 2:
locked resource.updating the balance ");
          }
        }
}};

t1.start();
t2.start();
  }
}
```

If all goes as planned, deadlock will occur, and the program will never exit.

Some ways to prevent deadlock

- Avoid deadlock by first locking **all the resources** in some predefined sequence at the start itself before starting on any thing critical.

```
void method1(){
synchronized(resource1){
synchronized(resource2){
…}}
}
```

```
void method2(){
synchronized(resource1){
synchronized(resource2){
…}}
}
```

Sequencing locking

The predefined sequence must be carefully thought out.
Will this code always work?

```
public void transferMoney(Account fromAccount,
Account toAccount, double amt) {
    synchronized (fromAccount) {
      synchronized (toAccount) {
       if (fromAccount.bal>amt) {
          fromAccount.debit(amt);
          toAccount.credit(amt);
        }
}    }  }
```

What if a transfer of money happens from account 111 to account 222 and at the same time a transfer of money happens from account 222 to account 111? Do see the deadlock occurring?

One solution to this is by locking the accounts in the increasing order. That is first lock 111 and then 222.

Exercise

- *Write a bank class that has an array of account objects. The method transfer allows transfer of money from one account to another account. Using the hints to avoid deadlocks from the previous slide implement the transfer method.*
- *Test the application by creating two threads that simultaneously transfers money form accounts 11111111 to account 22222222 and vice versa.*

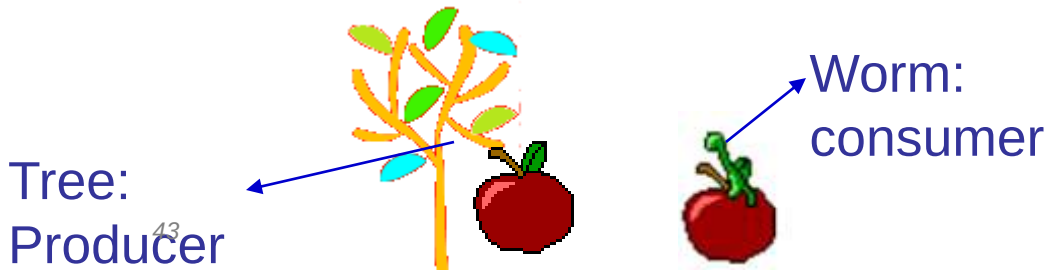
(30 mins)

Daemon threads

Go through Deamon threads topic in the reference material

Inter-thread communication

- Inter-thread communication is required when execution of one thread depends on another thread's task.
- In such case, the second thread intimates or notifies the first thread when it has finished some task that the first thread is waiting for.
- The best suited situation to understand this is a producer-consumer problem.
- A producer thread produces something which consumer thread consumes. A producer and consumer thread can run independently.



wait()and notify()

- The wait method of the **Object** class.

```
final void wait() throws InterruptedException

final void wait(long timeout) throws
InterruptedException

public final void wait(long timeout, int nanos)
throws InterruptedException
```

- **final void notify()**

  →Wakes up a single thread that is waiting on this object's lock. The choice is arbitrary .

  final void notifyAll()

  →Wakes up all threads that are waiting on this object's lock.

Exceptions thrown by `wait` and `notify`

- Overloaded **`wait()`** methods throw a checked exception which is **`InterruptedException.`**
- *So what does it mean when a method throws* **`InterruptedException?`**
- Overloaded **`wait() and notify()`** methods also throw an uncheck exception which is **`IllegalMonitorStateException.`**
- **`IllegalMonitorStateException`** is thrown if these methods are not called from a synchronized context.

Concurrency

- The concurrency utilities are contained in the **java.util.concurrent** package
- **java.util.concurrent** defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. It defines the following key features:
- • Synchronizers
- • Executors
- • Concurrent collections
- • The Fork/Join Framework

Synchronizers

- Offer high-level ways of synchronizing the interactions between multiple threads.
- Synchronization objects are supported by the **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger**, and **Phaser** classes. Collectively, they enable you to handle several formerly difficult synchronization situations with ease.

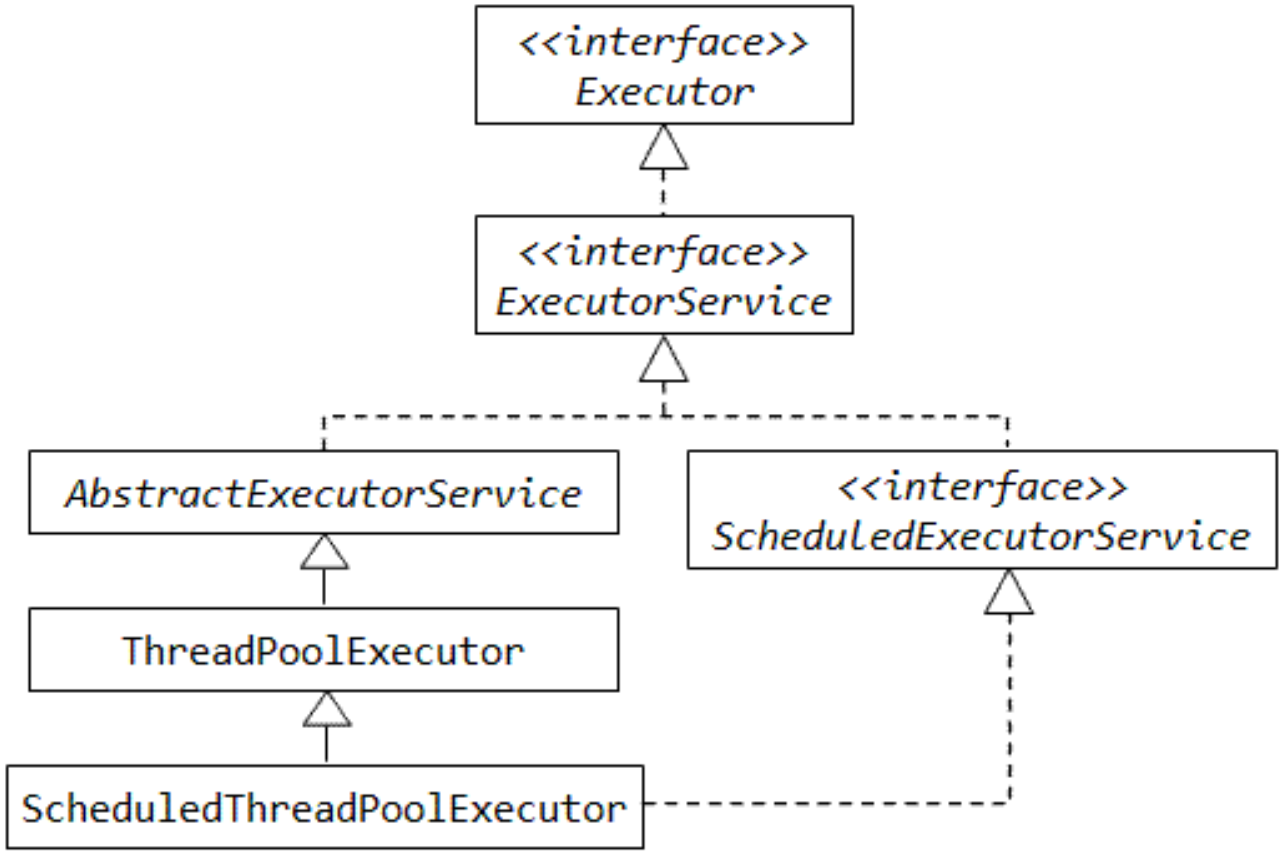
Concurrency – Use Executors and ThreadPools

- You can directly create and manage threads in the application by creating Thread objects.
- However, if you want to abstract away the low-level details of multi-threaded programming, you can make use of the Executor interface.

Executor

- Executor is an interface that declares only one method: void execute(Runnable). This may not look like a big interface by itself, but its derived classes (or interfaces), such as ExecutorService, ThreadPoolExecutor, and ForkJoinPool, support useful functionality.

Thread Pool, Executor, Callable/Future



Thread Pool

- To use a thread pool, you can use an implementation of the interface ExecutorService,:

- newFixedThreadPool
- newCachedThreadPool
- newSingleThreadedExecutor
- newScheduledThreadPool

contd

- newFixedThreadPool: this method returns a thread pool whose maximum size is *fixed*. It will create new threads as needed up to the maximum configured size. When the number of threads hits the maximum, the thread pool will maintain the size constant.
- newCachedThreadPool: this method returns an *unbounded* thread pool, that is a thread pool without a maximum size. However, this kind of thread pool will tear down unused thread when the load reduces

contd

- newSingleThreadedExecutor: this method returns an executor that guarantees that tasks will be executed in a single thread.
- newScheduledThreadPool: this method returns a fixed size thread pool that supports delayed and timed task execution.

Steps

- Write your worker thread class which implements Runnable interface. The run() method specifies the behavior of the running thread.
- Create a thread pool (ExecutorService) using one the factory methods provided by the Executors class.
- The thread pool could have a single thread, a fixed number of threads, or an unbounded number of threads.

contd

- Create instances of your worker thread class.
  Use execute(Runnable r) method of the thread pool to add a Runnable task into the thread pool.

- The task will be scheduled and executes if there is an available thread in the pool.

Interface java.util.concurrent.Executor

- An Executor object can execute Runnable tasks submitted. The interface declares an abstract method:
- public void execute(Runnable r) It executes the given task at some time in the future. The task may be executed in a new thread, in a thread pool, or in the calling thread, depending on the implementation of Executor (e.g. single thread or thread pool)

Interface java.util.concurrent.ExecutorService

- Interface ExecutorService declares many abstract methods. The important ones are:
- public void shutdown(); // Initiates an orderly shutdown of the thread pool. // The previously executed/submitted tasks are allowed to complete, // but no new tasks will be scheduled. public <T> Future<T> submit(Callable<T> task); // Submit or schedule the callable task for execution, which returns a Future object.

Class java.util.concurrent.Executors

- The class Executors provides factory methods for creating Executor object. For example:
- static ExecutorService newSingleThreadExecutor() static ExecutorService newFixedThreadPool(int nThreads) static ExecutorService newCachedThreadPool() static ScheduledExecutorService newSingleThreadScheduledExecutor() static ScheduledExecutorService newScheduledThreadPool(int size)

Callable and Future

- A Callable is similar to a Runnable. However, Callable provides a way to return a result or Exception to the thread that spin this Callable. Callable declares an abstract method call() (instead of run() in the Runnable).
- public V call() // Call() returns a result of type <V>, or throws an exception if unable to do so.

Contd.

- In the thread pool, instead of using execute(Runnable r), you use submit(Callable r), which returns a Future<V> object (declared in the ExecutorService interface).
- When the result is required, you can retrieve using get() method on the Future object. If the result is ready, it is returned, otherwise, the calling thread is blocked until the result is available.
- The interface Future<V> declares the following abstract methods:
- V get() // wait if necessary, retrieve result V get(long timeout, TimeUnit unit) boolean cancel(boolean mayInterruptIfRunning) boolean isCancelled() boolean isDone() // return true if this task completed

Exercise

- *Consider the following scenario. Whenever a hen lays an egg its owner sells the egg to a shop. In the last 4 months the owner has gained Rs. 100 by selling eggs in the rate of Rs.2 per egg. Display the following messages*

  Hen Laid the Egg – 1

  Owner gained Rs 2

  Hen Laid the Egg – 2

  Owner gained Rs 4

  …

  …

  So on.

(45 mins)

Summary

- Thread in java is created using native methods which in turn create threads based on the OS. Java SE provides 2 classes for working with thread called Thread and ThreadGroup.
- A deamon thread is a special type of thread that runs in background.
- The two ways to create threads are by extending Thread class and by creating Thread object and passing Runnable instance.
- sleep() causes the currently executing thread to sleep for the minimum of specified number of milliseconds.
- A thread can be interrupted while sleeping or waiting by calling interrupt().
- A thread enters a synchronized block of code only if no other thread is executing any other a synchronized block of code on that object.
- Inter-thread communication is required when execution of one thread depends on another thread's task.

Contd.

- Java.util.concurrency package provides high level concurrent utility classes
- Synchronisers : Semaphore, countdownlatch, cyclic barrier, exchanger and phaser
- Thread pool executors control and manage thread execution