Hexaware TECHNOLOGIES

SDET Training Program

Threads

Thread in a Java program

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

Prints : main

Threads

- Sun defines a thread as a single sequential flow of control within a program.
- Threads are the code sequence that execute within a process.
- It is sometimes referred to as an execution context or a lightweight process.
- Thread based multitasking environments allow a single program to perform two or more tasks simultaneously.
- The thread in java is, in-fact, a realization of OS level thread.

Important Roles of Threads

- 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.
- Do other things while waiting for slow I/O operation.
- Animations

Creating Threads

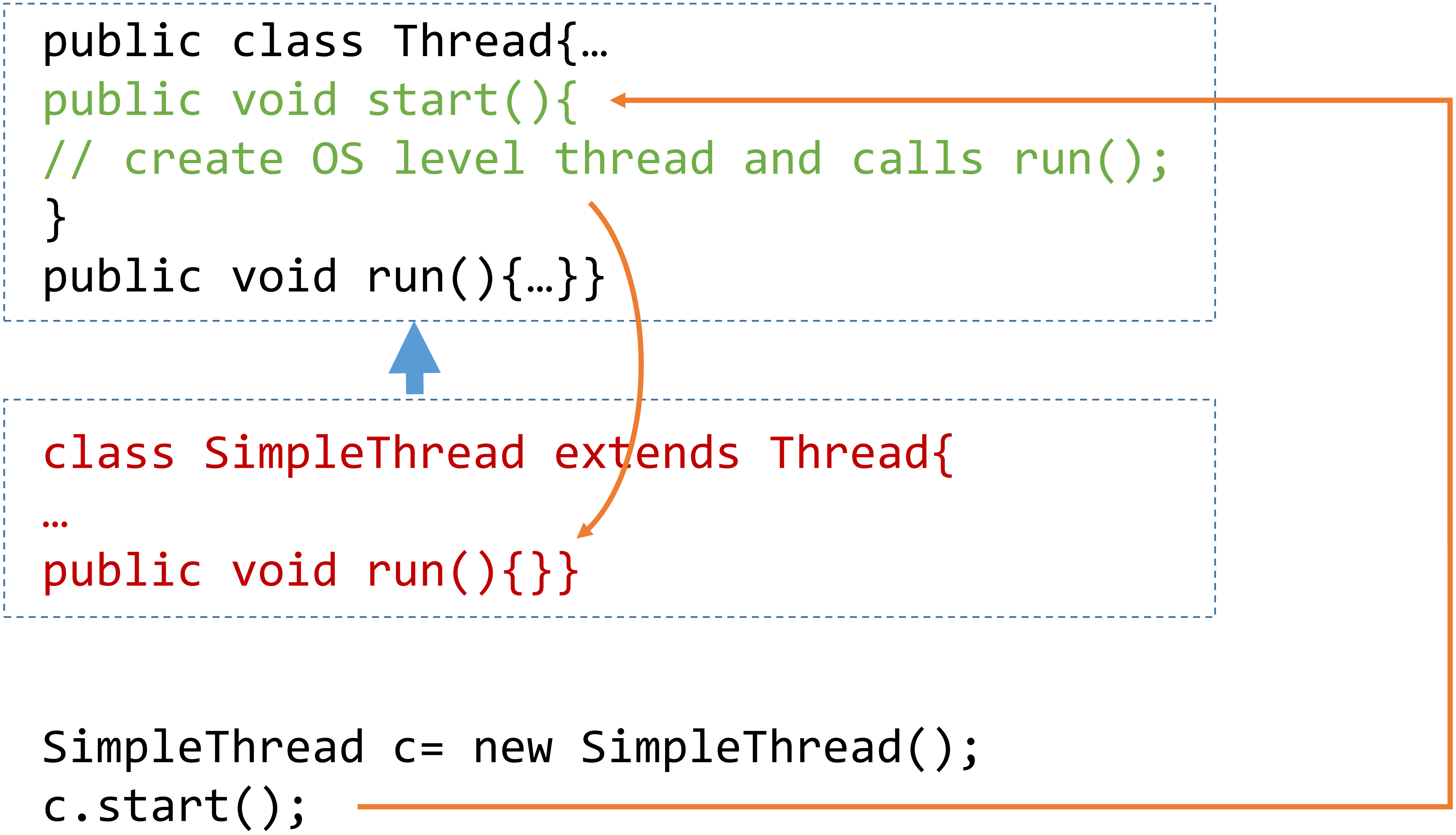
or `java.lang.Thread`

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

```
SimpleThread t= new SimpleThread();
t.start();
```

Calls run() method

- We override the `run()` method and put the code that needs to be executed when the thread runs, in the run() method.
- To call run method we call `start()` !



```
public class Thread{…
public void start(){
// create OS level thread and calls run();
}
public void run(){…}}
```

```
class SimpleThread extends Thread{
…
public void run(){}}
```

```
SimpleThread c= new SimpleThread();
c.start();
```

Let’s Do a Complete Example

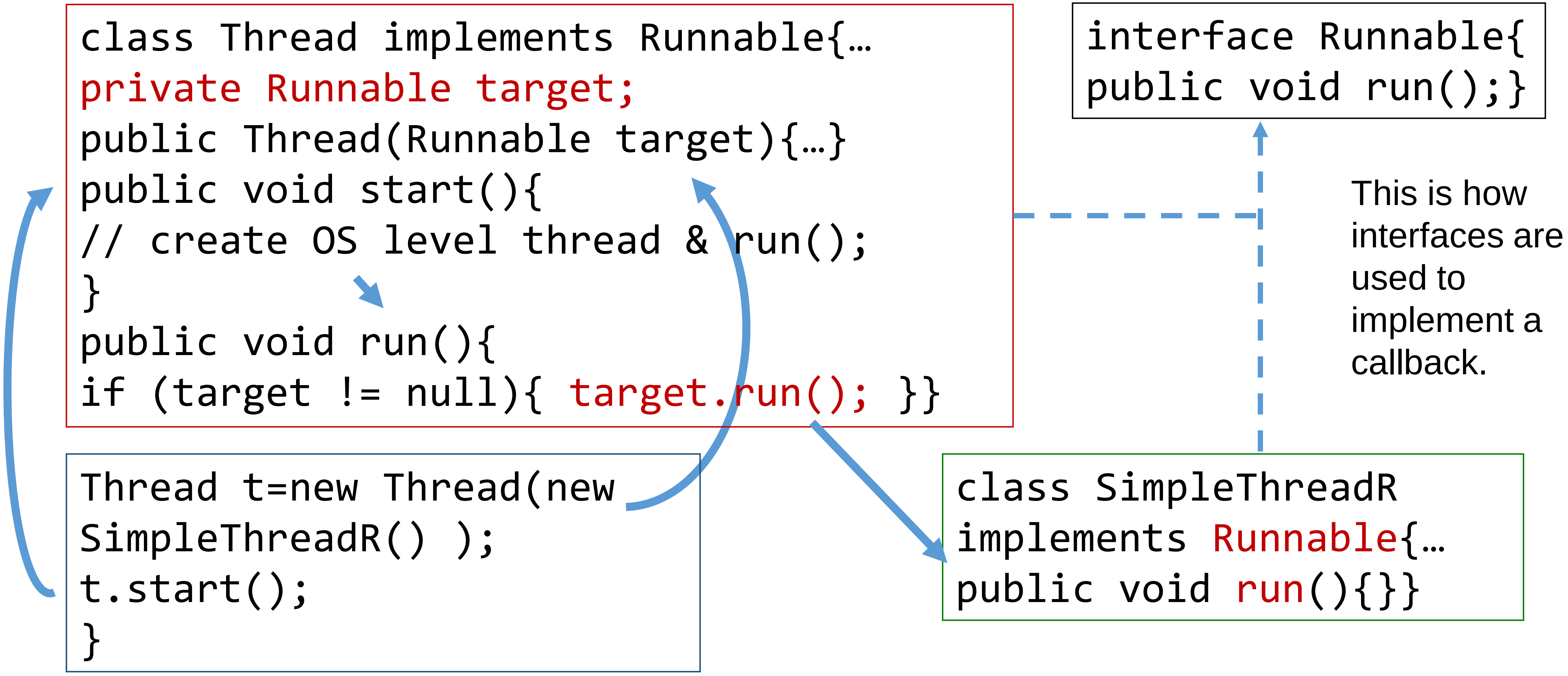
Another Way to Work with Threads

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

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

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

Callback Working Behind the Scenes



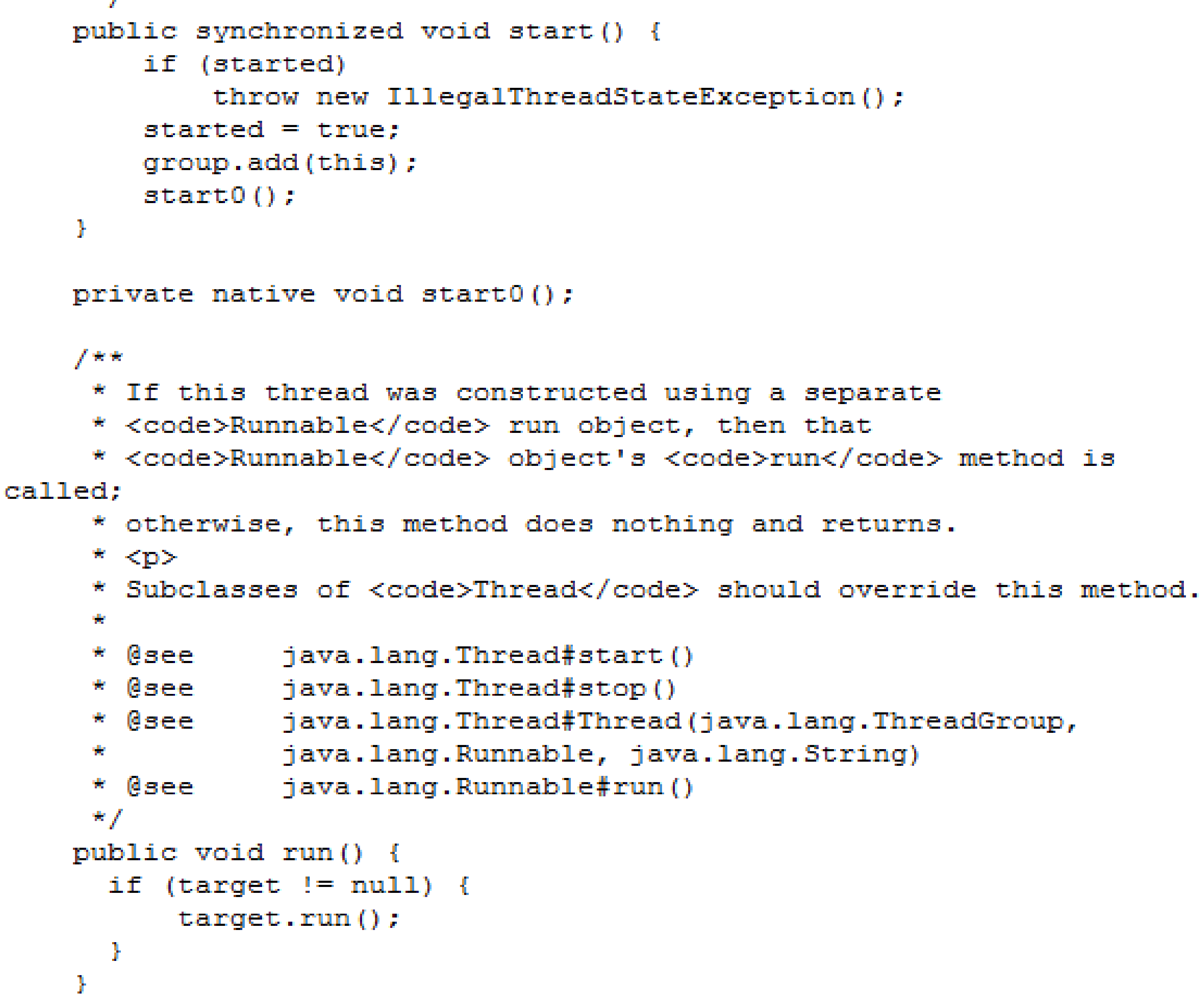
```
class Thread implements Runnable{…
private Runnable target;
public Thread(Runnable target){…}
public void start(){
// create OS level thread & run();
}
public void run(){
if (target != null){ target.run(); }}
```

```
interface Runnable{
public void run();}
```

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

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

This is how interfaces are used to implement a callback.



```
    public synchronized void start() {
        if (started)
            throw new IllegalThreadStateException();
        started = true;
        group.add(this);
        start0();
    }

    private native void start0();

    /**
     * If this thread was constructed using a separate
     * <code>Runnable</code> run object, then that
     * <code>Runnable</code> object's <code>run</code> method is
called;
     * otherwise, this method does nothing and returns.
     * <p>
     * Subclasses of <code>Thread</code> should override this method.
     *
     * @see     java.lang.Thread#start()
     * @see     java.lang.Thread#stop()
     * @see     java.lang.Thread#Thread(java.lang.ThreadGroup,
     *          java.lang.Runnable, java.lang.String)
     * @see     java.lang.Runnable#run()
     */
    public void run() {
      if (target != null) {
          target.run();
      }
    }
```

Naming Threads

Using Constructors:

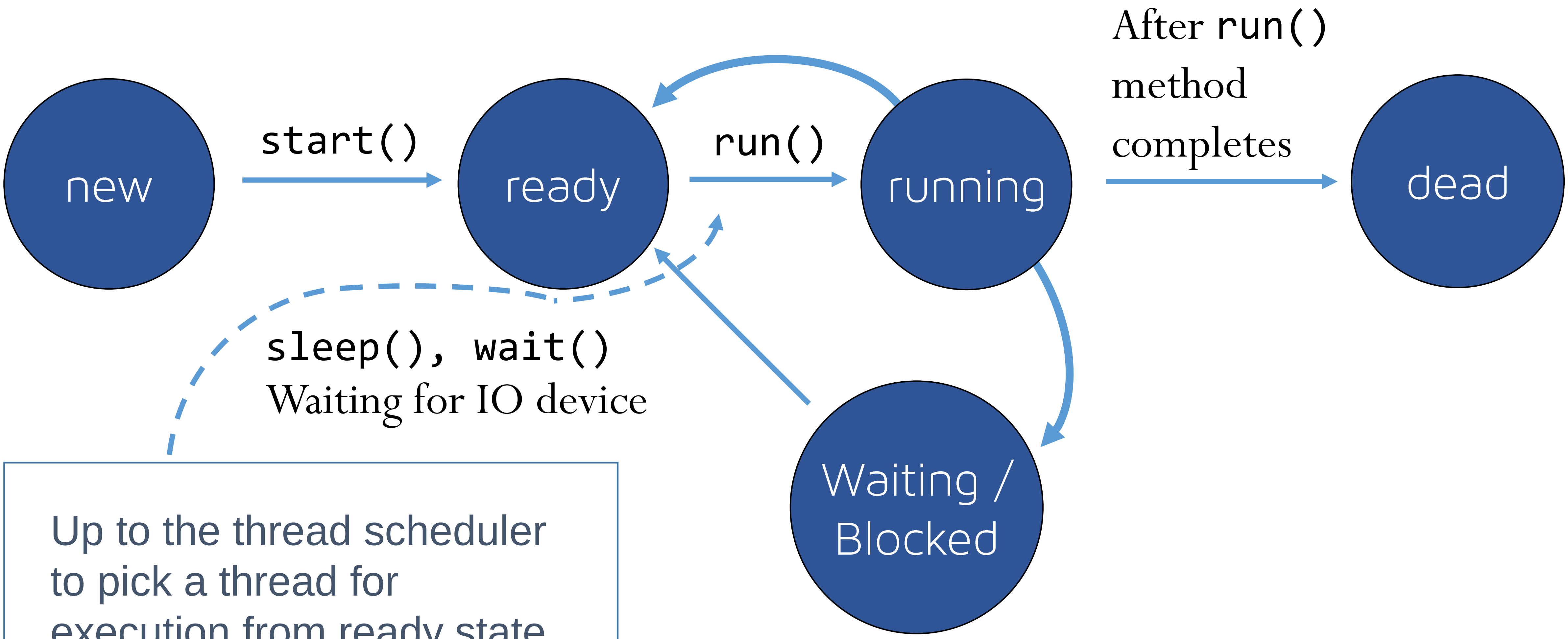
- `Thread(Runnable target, String name)`

Methods:

- `final void setName(String name)`
- `final String getName()`

- `Thread.currentThread().getName();`
- `static Thread currentThread()` returns the current thread
- The default name of a user defined thread is 'Thread-0' for the first thread created, 'Thread-1' for the second and so on.

Lifecycle



Up to the thread scheduler to pick a thread for execution from ready state.

`sleep()`

- `static void sleep(long millis) throws InterruptedException`

- `static void sleep(long millis, int nanos) throws InterruptedException`

`sleep()` causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

```
public class Appear implements Runnable{
   char c[]={‘H',‘E',‘L',‘L‘,’O’};

public void run() {
   int i=0;
   try{
   while (i<5){
   System.out.print(c[i++]);
   Thread.sleep(1000);
   }
   }catch(InterruptedException e){}
   }
public static void main(String str[]){
Thread t =new Thread(new Appear());
t.start();
}}
```

Can you guess what this code does?

join()

- `final void join() throws InterruptedException`
- `final void join(long millis) throws InterruptedException`
- `final void join(long millis, int nanos) throws InterruptedException`

When a thread calls `join()` on another thread instance, the caller thread will wait till the called thread finishes execution.

Let’s Do a Complete Example

+

Join

EXERCISE

- Ask the user a question
- Start another thread that will increment count after every 1 sec.
- Moment the user to answers, interrupt the thread and
- Display count and correctness of the answer
- Use `Runnable` interface.

Thread Priorities

```
final void setPriority(int newPriority) throws
   IllegalArgumentException
```

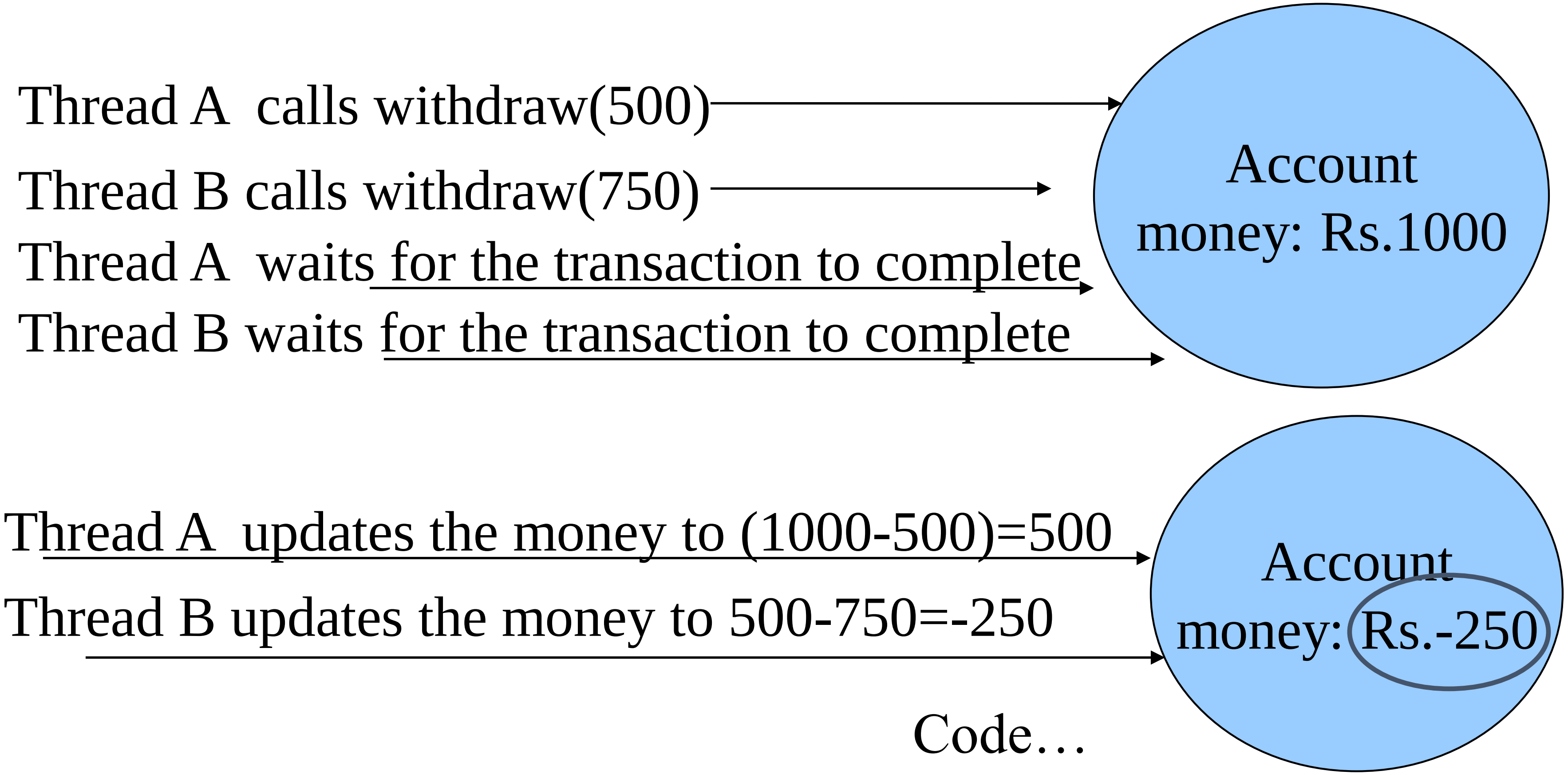
```
final int getPriority()
```

Static constants to set priorities:

```
Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5): default
Thread.MAX_PRIORITY (10)
```

- This is the only way to influence the scheduler’s decision as to the order of thread execution.
- The new threads inherit the priority from the thread that created it.

Synchronization



Thread A calls withdraw(500)

Thread B calls withdraw(750)

Thread A waits for the transaction to complete

Thread B waits for the transaction to complete

Account
money: Rs.1000

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

Thread B updates the money to 500-750=-250

Account
money: Rs.-250

Code…

```
class Account{
private int money;
Account(int amt){
//get amt from database
money=amt;}
void withdraw(int amt){
if(amt<money){

try{
 Thread.sleep(1000);
 money=money-amt;
 }catch(Exception e){}

System.out.println("Received "+ amt  +" by " +
Thread.currentThread().getName());
}
```

→ simulating time to connect to other systems and performing IO operation

```
else          → If amt not available
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
```

If two threads access the same object and each calls a method that changes the state of that object then data corruption can result. This is called race condition.

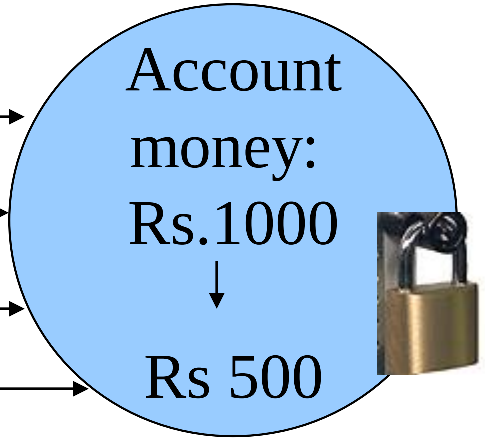
Solution to the Account problem

Thread A locks the object and calls withdraw(500)

Thread B waits for A to release the lock

Thread A waits for the transaction to complete

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



Account money: Rs.1000

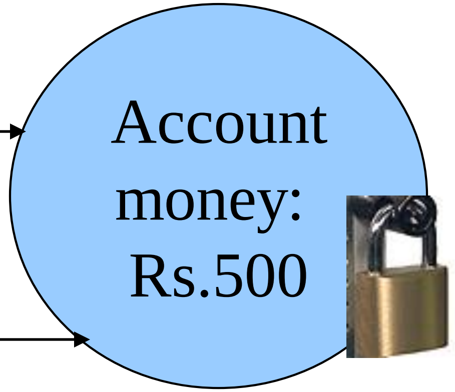
↓

Rs 500

Thread B locks the object and calls withdraw(750)

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

Thread B releases the lock.



Account money: Rs.500

- Approach 1: automatically when thread calls synchronized method.

Add synchronized keyword to withdraw and other critical methods of the Account class.

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

- Approach 2: locking an object explicitly

Change the run method of ThreadTest class to

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

Deadlock

- Thread 1 locks resource 1
- Thread 2 locks resource 2
- Thread 1 waits for resource 2 to be released
- Thread 2 waits for resource 1 to be released
- Ends up in a DEADLOCK

```
public class Lock{
 public static void main(String[] args) {
  final Account resource1 = "resource1";
  final Account resource2 = "resource2";
  // 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) { }
   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() {
       synchronized (resource2) {
           System.out.println("Thread 2: locked resource.
           updating the balance ");
           try {
           Thread.sleep(50);
           } catch (InterruptedException e) {
           }

           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.

What could you have done to avoid this deadlock?

Deadlock Prevention

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){
…}}
}
```

Money Transfer

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

The solution suggested in the previous slide fails in this case.