

5COSC019W – Object Oriented Programming Week 10

Dr. Barbara Villarini

b.villarini@westminster.ac.uk



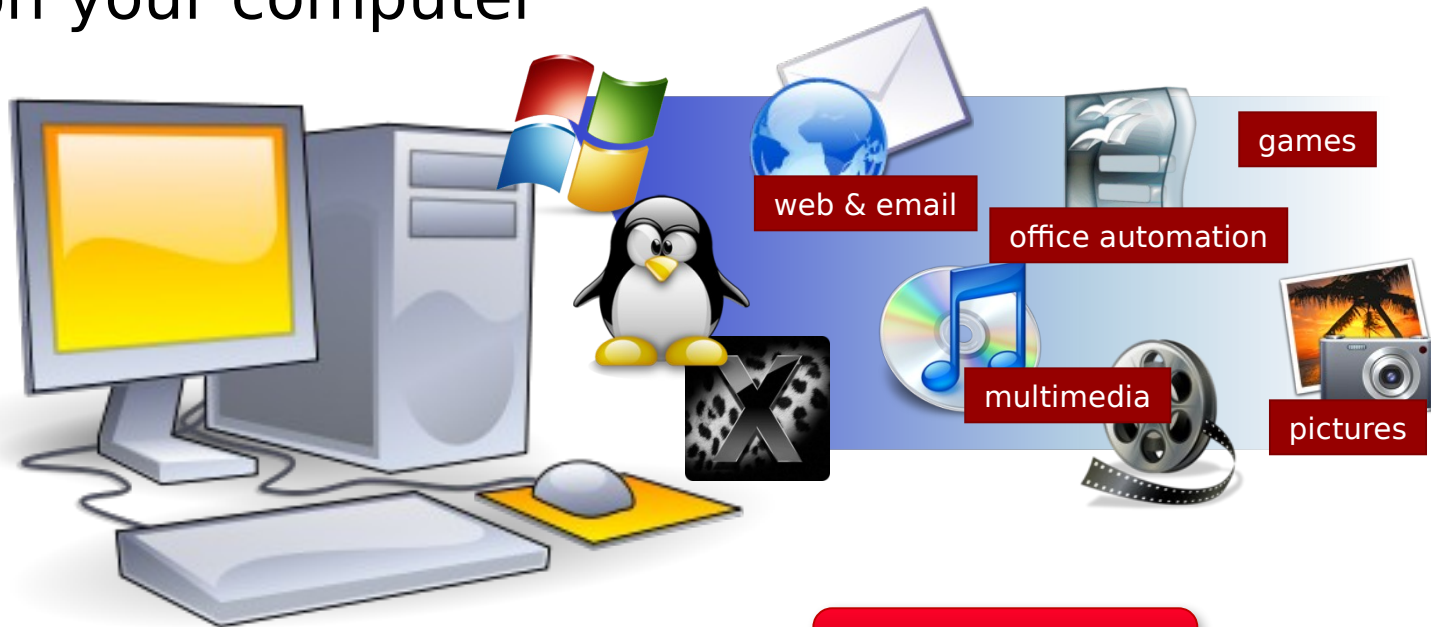
Overview

- Multithreading
- Threads in Java
- Thread class and Runnable interface
- Concurrency
- Synchronization



Threaded Applications

- Modern Systems
 - Multiple applications run concurrently!
 - This means that... there are multiple processes on your computer



Multitasking



Threaded Applications

- Modern Systems
 - Applications perform many tasks at once!
 - This means that... there are multiple threads within a single process.





Thread

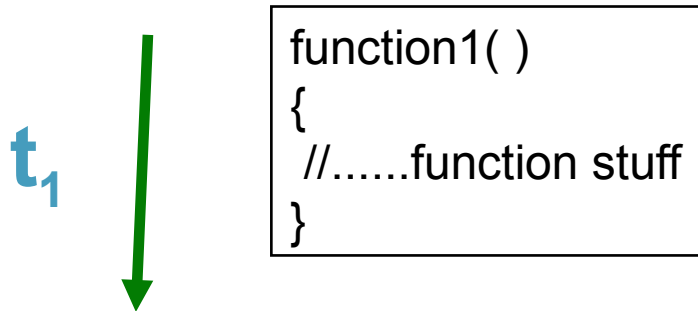
- Thread: single sequential flow of control within a program
- Single-threaded program can handle one task at any time.
- Multitasking allows single processor to run several concurrent threads.
- Most modern operating systems support multitasking.

Serial vs. Parallel





Serial vs. Parallel in Computing

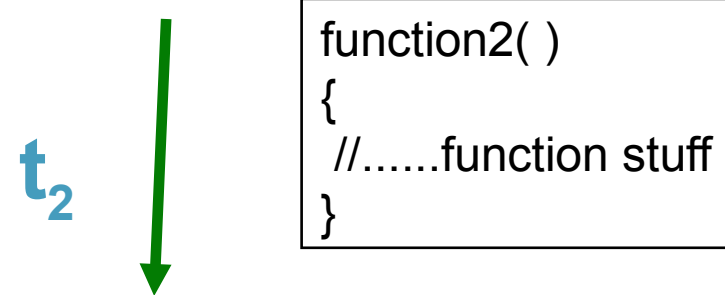


Serial Machine

`function1 ():`
`function2 ():`

- Single CPU

Time : add (t_1 , t_2)



Parallel Machine : MPP

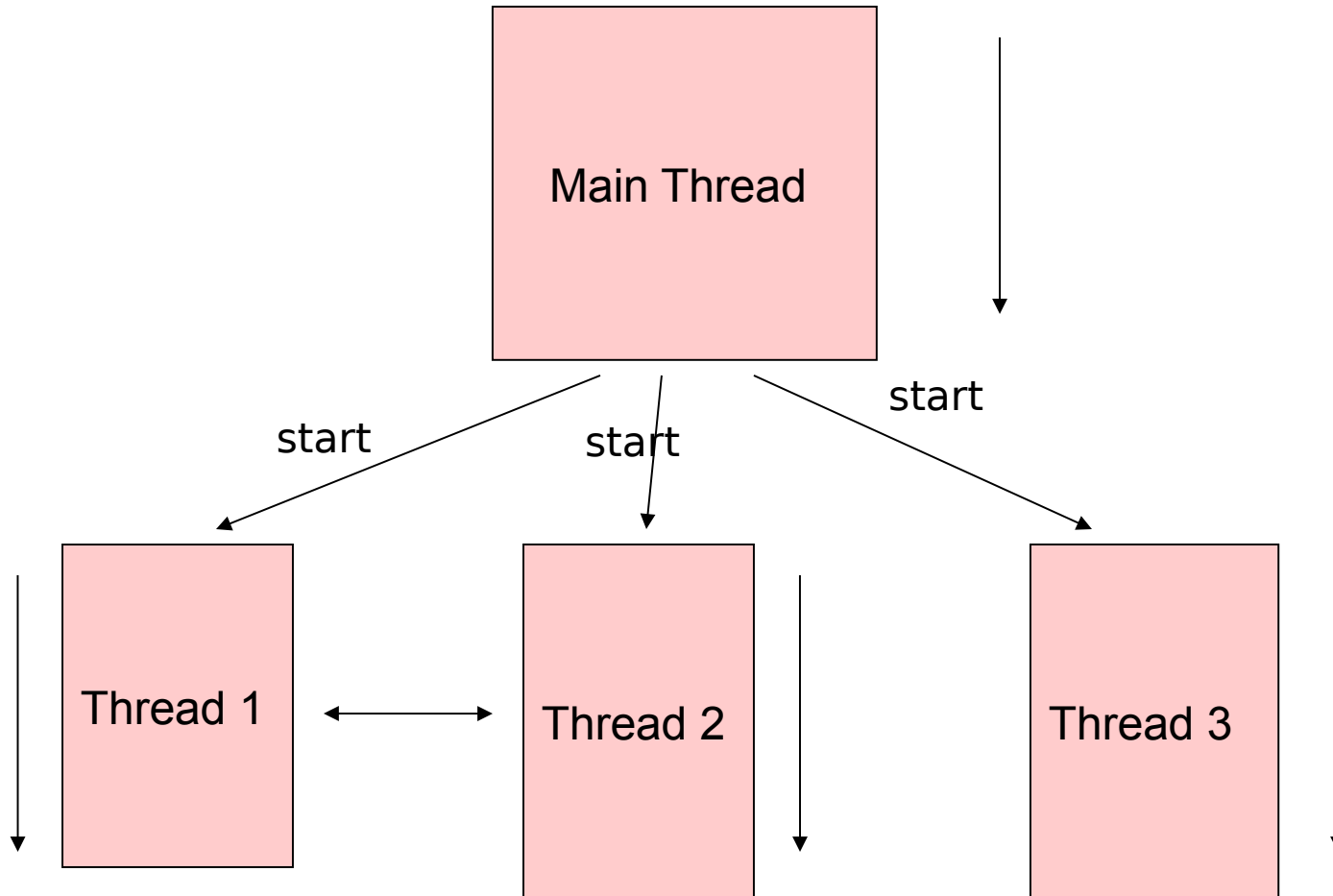
`function1() || function2 ()`

- massively parallel system containing thousands of CPUs

Time : max (t_1 , t_2)

A multithread program

threads are light-weight processes within a process

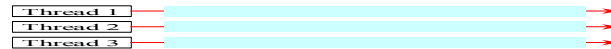


Threads may switch or exchange data/results

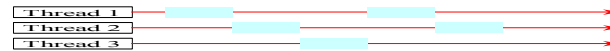


Threads Concept

Multiple
threads on
multiple
CPUs



Multiple
threads
sharing a
single CPU





Summing Up

- A Thread is a piece of code that runs in concurrent with other threads.
- Each thread is a statically ordered sequence of instructions.
- Threads are used to express concurrency on both single and multiprocessors machines.
- Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.



Threads in Java

- Java has built in support for Multithreading
- Synchronization
- Thread Scheduling
- Inter-Thread Communication:

currentThread

start

setPriority

yield

run

getPriority

sleep

stop

suspend

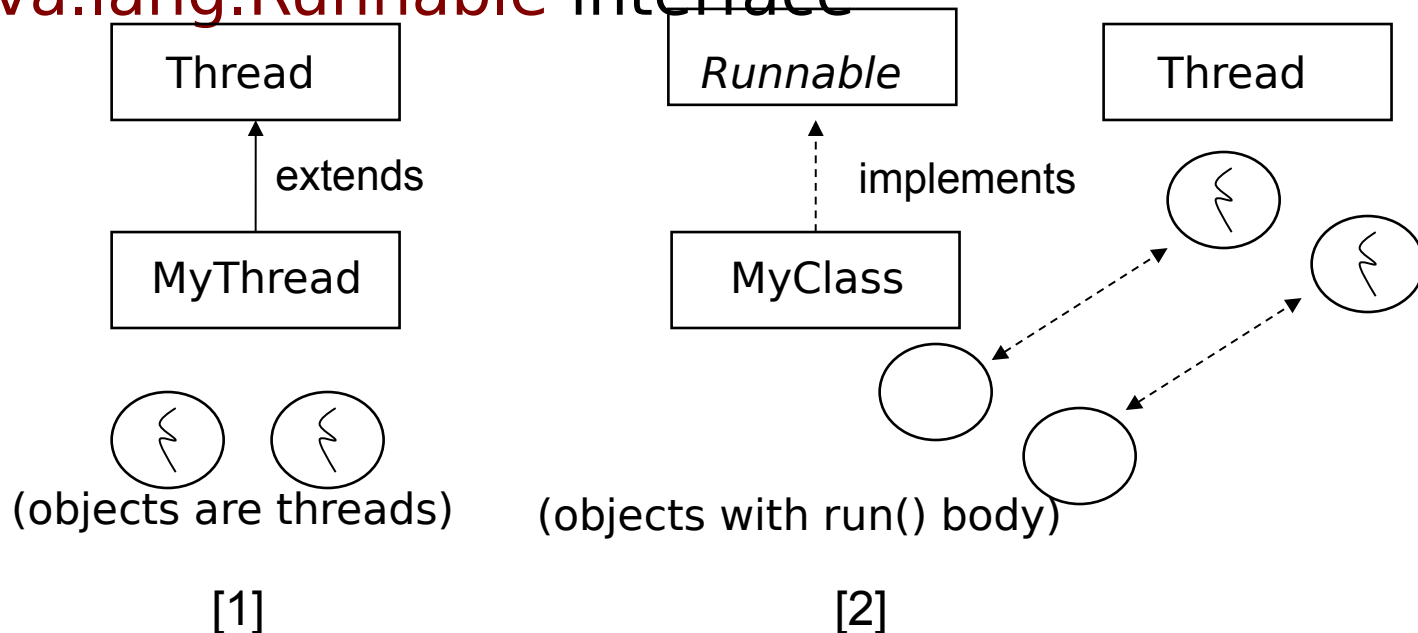
resume

- Java Garbage Collector is a low-priority thread.



2 Threading Mechanisms

1. Create a class that extends the `java.lang.Thread` class
2. Create a class that implements the `java.lang.Runnable` interface





1st Method: Extending Thread Class

- Create a class by extending Thread class and override **run() method**:

```
class MyThread extends Thread
{
    public void run()
    {
        // thread body of execution
    }
}
```

- Create a thread:
`MyThread thr1 = new MyThread();`
- Start Execution of threads:
`thr1.start();`



Example

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println(" this thread is  
                             running ... ");  
    }  
}  
  
class ThreadEx1 {  
    public static void main(String [] args ) {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```



2nd method: Threads by implementing Runnable interface

- Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
    .....
    public void run()
    {
        // thread body of execution
    }
}
```

- Creating Object:
`MyThread myObject = new MyThread();`
- Creating Thread Object:
`Thread thr1 = new Thread(myObject);`
- Start Execution:
`thr1.start();`



Example

```
class MyThread implements Runnable {
    public void run() {
        System.out.println(" this thread is
                           running ... ");
    }
}

class ThreadEx2 {
    public static void main(String [] args ) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```




Overview of Thread Methods

- Thread-related methods
 - See API for more details (especially exceptions)
 - **run**
 - Does “work” of a thread – What does this mean?
 - Can be overridden in subclass of **Thread** or in **Runnable** object
 - **start**
 - Launches thread, then returns to caller
 - Calls **run**
 - Error to call **start** twice for same thread

Thread States: Life Cycle of a Thread

- Born state
 - Thread just created
 - When **start** called, enters ready state
- Ready state (runnable state)
 - Highest-priority ready thread enters running state
- Running state
 - System assigns processor to thread (thread begins executing)
 - When **run** completes or terminates, enters dead state
- Dead state
 - Thread marked to be removed by system
 - Entered when **run** terminates or throws uncaught exception

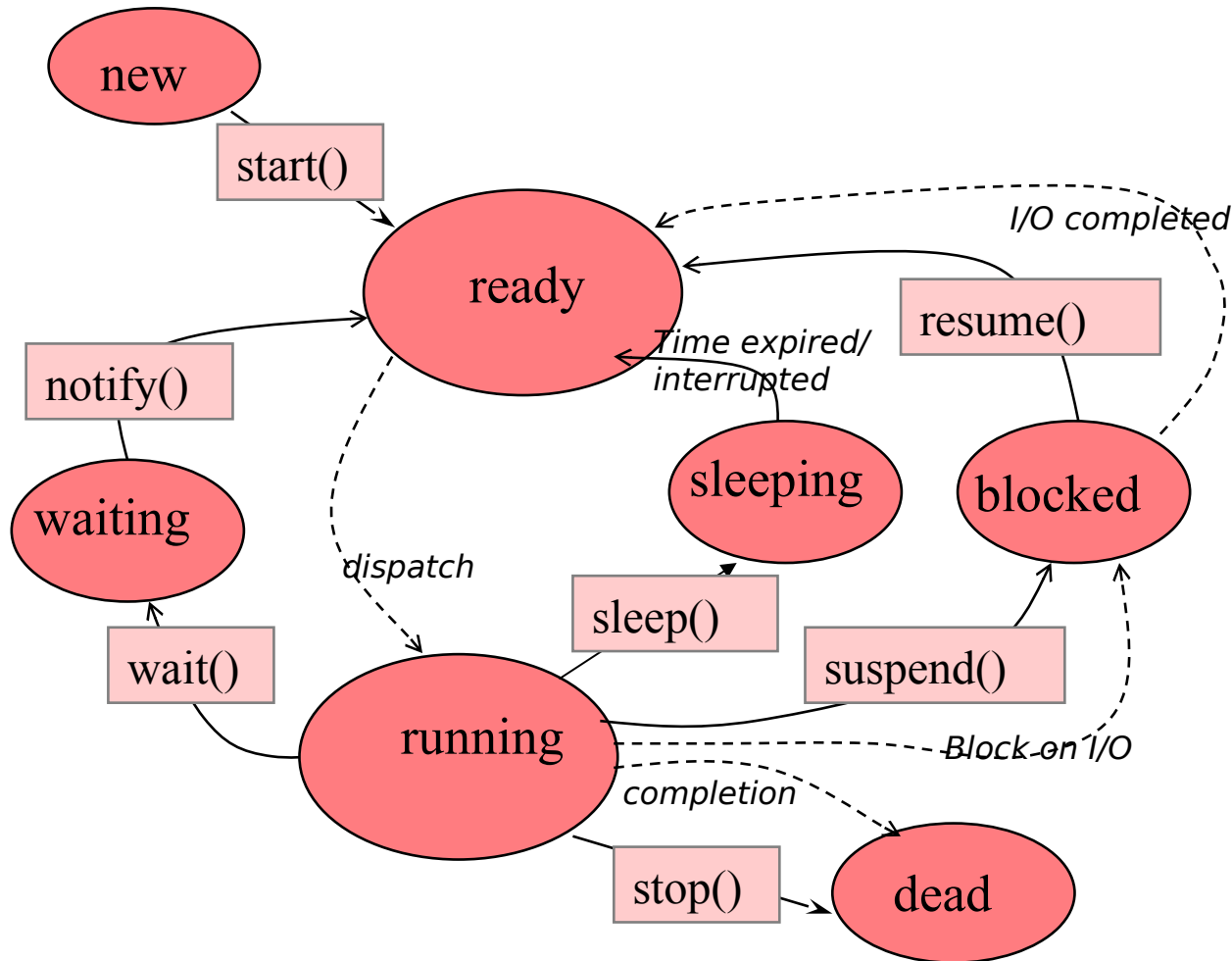


Other thread states

- Blocked state
 - Entered from running state
 - Blocked thread cannot use processor, even if available
 - Common reason for blocked state - waiting on I/O request
- Sleeping state
 - Entered when `sleep` method called
 - Cannot use processor
 - Enters ready state after sleep time expires
- Waiting state
 - Entered when `wait` called in an object thread
 - One waiting thread becomes ready when object calls `notify`
 - `notifyAll` - all waiting threads become ready



Life Cycle of Thread



A Program with Three Java Threads

- Implement a program that creates 3 threads

Three threads example

```
class A extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("\t From ThreadA: i= "+i);
        }
        System.out.println("Exit from A");
    }
}

class B extends Thread
{
    public void run()
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("\t From ThreadB: j= "+j);
        }
        System.out.println("Exit from B");
    }
}
```

Three threads example

```
class C extends Thread
{
    public void run()
    {
        for(int k=1;k<=5;k++)
        {
            System.out.println("\t From ThreadC: k= "+k);
        }

        System.out.println("Exit from C");
    }
}

class ThreadTest
{
    public static void main(String args[])
    {
        new A().start();
        new B().start();
        new C().start();
    }
}
```



Consideration

- The three threads are not exactly interleaved.
- The thread scheduler gives no guarantee about the order in which threads are executed.
- Each thread runs for a small amount of time called **Time Slice**
- The order in which each thread gains control is somehow **random**



Terminating Threads

- In Java 1.1, the Thread class had a stop() method
 - One thread could terminate another by invoking its stop() method.
 - However, using stop() could lead to deadlocks
 - **The stop() method is now deprecated.** DO NOT use the stop method to terminate a thread
- The correct way to stop a thread is to have the run method terminate
 - Add a boolean variable which indicates whether the thread should continue or not
 - Provide a set method for that variable which can be invoked by another thread

Example



```
public class ThreadToStop extends Thread{

    private boolean exit = false;

    public void run(){
        while(!exit){
            System.out.println("Thrad is running...");
        }
        System.out.println("Thread is stopped!");
    }

    public void stopThread() {
        exit = true;
    }
}
```

Set the
variable from
outside the
program

```
public static void main(String[] args) throws
InterruptedException {

    ThreadToStop thread = new ThreadToStop();
    thread.start();

    Thread.sleep(1000);

    thread.stopThread();
}
```



Terminating threads – Interrupting method

- To notify a thread that it should clean up and terminate you can use the `interrupt` method

```
t.interrupt();
```

- This method set a Boolean variable in the thread data structure
- In your run method you can check for interruptions for each iteration

```
public void run(){
    for (int i = 0; i<=Repetition && !Thread.interrupted(); i++)
    {
        // Do work
    }
    // Clean up
}
```

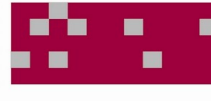


Thread Priorities

Each thread is assigned a default priority of `Thread.NORM_PRIORITY` (default priority). You can reset the priority using `setPriority(int priority)`.

Some constants for priorities include

`Thread.MIN_PRIORITY` `Thread.MAX_PRIORITY`
`Thread.NORM_PRIORITY`



Yield() and sleep()

- Sometimes a thread can determine that it has nothing to do
 - Sometimes the system can determine this. ie. waiting for I/O
- When a thread has nothing to do, it should not use CPU
 - This is called a busy-wait.
 - Threads in busy-wait are busy using up the CPU doing nothing.
 - Often, threads in busy-wait are continually checking a flag to see if there is anything to do.
- It is worthwhile to run a CPU monitor program on your desktop
 - You can see that a thread is in busy-wait when the CPU monitor goes up (usually to 100%), but the application doesn't seem to be doing anything.
- Threads in busy-wait should be moved from the Run queue to the Wait queue so that they do not hog the CPU
 - Use yield() or sleep(time)
 - Yield simply tells the scheduler to schedule another thread
 - Sleep guarantees that this thread will remain in the wait queue for the specified number of milliseconds.



Potential risk when a thread is sleeping

- A thread might sleep for so long that is no longer useful and should be terminated.
- We just saw that to terminate the thread we interrupt it.
- When a sleeping thread is interrupted an `InterruptedException` is generated
- **You need to catch this exception!**

```
public void run() {  
    try{  
        Task Statements  
    }  
    catch (InterruptedException exception)  
    {  
    }  
    Clean up if necessary  
}
```



Concurrent Access to Data

- Concurrent access to data can lead to data integrity problems
 - Specifically, if two sources attempt to update the same data at the same time, the result of the data can be undefined.
 - The outcome is determined by how the scheduler schedules the two sources.
 - Since the scheduler's activities cannot be predicted, the outcome cannot be predicted
- Databases deal with this mechanism through "locking"
 - If a source is going to update a table or record, it can lock the table or record until such time that the data has been successfully updated.
 - While locked, all access is blocked except to the source which holds the lock.
- Java has the equivalent mechanism. It is called synchronization



Read/Write problem

- If one thread tries to read the data and another thread tries to update the same data leads to inconsistent state for the shared data.
- This can be prevented by synchronizing access to the data via Java **synchronized** keyword

```
public synchronized void update() {  
    . . .  
}
```




Read/Write Problem example

- Consider an example of a bank offering online access to its customers to perform transactions on their accounts

```
public class InternetBankingSystem {  
    public static void main(String [] args ) {  
        Account accountObject = new Account(100);  
        new Thread(new DepositThread(accountObject,30)).start();  
        new Thread(new DepositThread(accountObject,20)).start();  
        new Thread(new DepositThread(accountObject,10)).start();  
        new Thread(new WithdrawThread(accountObject,30)).start();  
        new Thread(new WithdrawThread(accountObject,50)).start();  
        new Thread(new WithdrawThread(accountObject,20)).start();  
    } // end main()  
}
```

Example – withdraw thread

```
public class WithdrawThread implements Runnable {  
  
    private Account account;  
    private double amount;  
  
    public WithdrawThread(Account account, double amount) {  
        this.account = account;  
        this.amount = amount;  
    }  
  
    public void run() {  
        //make a withdraw  
        account.withdraw(amount);  
    }  
}
```



Example – Deposit thread

```
public class DepositThread implements Runnable {  
  
    private Account account;  
    private double amount;  
  
    public DepositThread(Account account, double amount) {  
        this.account = account;  
        this.amount = amount;  
    }  
  
    public void run() {  
        //make a deposit  
        account.deposit(amount);  
    }  
}
```



Example – Class Account

```
public class Account {  
    private double balance = 0;  
  
    public Account(double balance) {  
        this.balance = balance;  
    }  
  
    public void deposit(double amount) {  
        if (amount < 0) {  
            System.out.println("Can't deposit.");  
        }  
        else{ this.balance += amount;  
            System.out.println("Deposit" + amount + " in thread"  
                + Thread.currentThread().getId() + ", balance is " + balance);  
        }  
    }  
  
    public void withdraw(double amount) {  
        if (amount < 0 || amount > this.balance) {
```



Output

- Without synchronization an unpredictable behavior will occur.
- Run may generate the following printout in the system console:

```
Withdraw 30.0 in thread 11, balance is 60.0  
Deposit 20.0 in thread 9, balance is 60.0  
Withdraw 50.0 in thread 12, balance is 60.0  
Deposit 10.0 in thread 10, balance is 60.0  
Withdraw 20.0 in thread 13, balance is 60.0  
Deposit 30.0 in thread 8, balance is 60.0
```

How we solve this problem?

- How will we ensure that only one class can access the data at one time ?
- We can use the **Synchronized** keyword

Example



```
public class Account {  
    private double balance = 0;  
  
    public Account(double balance) {  
        this.balance = balance;  
    }  
  
    public synchronized void deposit(double amount) {  
        if (amount < 0) {  
            System.out.println ("Can't deposit.");  
        }  
  
        this.balance += amount;  
  
        System.out.println("Deposit" + amount + " in thread"  
            + Thread.currentThread().getId() + ", balance is " + balance);  
    }  
  
    public synchronized void withdraw(double amount) {  
        if (amount < 0 || amount > this.balance) {  
            System.out.println ("Can't withdraw.");  
        }  
    }  
}
```



Output

- The synchronized method will ensure that only one class can access the data at one time, other operations have to wait until the first operation finishes
- Simply run the InternetBankingSystem program, the following results should be shown on the system console:

```
Deposit 30.0 in thread 8,new balance is 130.0  
Withdraw 50.0 in thread 12,new balance is 80.0  
Deposit 10.0 in thread 10,new balance is 90.0  
Withdraw 20.0 in thread 13,new balance is 70.0  
Withdraw 30.0 in thread 11,new balance is 40.0  
Deposit 20.0 in thread 9,new balance is 60.0
```




Synchronizing Instance Methods and Static Methods

- A synchronized method acquires a lock before it executes.
- **Instance method: the lock is on the object for which it was invoked.**
- **Static method: the lock is on the class.**
- If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired, then the method is executed, and finally the lock is released.
- Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.



Synchronizing Statements

- A *synchronized block* can be used to acquire a lock on any object, when executing a block of code.

```
synchronized (expr) {  
    statements;  
}
```

- `expr` must evaluate to an object reference.
- If the object is already locked by another thread, the thread is blocked until the lock is released.
- When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.



Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- You can use locks explicitly to obtain more control for coordinating threads.
- A lock is an instance of the **Lock** interface, which declares the methods for acquiring and releasing locks.



Lock implementation

- **ReentrantLock**: concrete implementation of Lock for creating mutually exclusive locks.

```
Lock balanceLock = new ReentrantLock();
```

```
...
```

```
balanceLock.lock();
```

```
Manipulate the shared code here
```

```
balanceLock.unlock();
```

```
...
```

Example using locks



```
public class Account {  
    private double balance = 0;  
    private Lock balanceLock;
```

Declare the lock

```
    public Account(double balance) {  
        this.balance = balance;  
        balanceLock = new ReentrantLock();  
    }
```

Instantiate the lock

```
    public void deposit(double amount) {  
        if (amount < 0) {  
            System.out.println ("Can't deposit.");  
        }  
        balanceLock.lock();  
        this.balance += amount;  
        System.out.println("Deposit" + amount + " in thread"  
            + Thread.currentThread().getId() + ", balance is " + balance);  
        balanceLock.unlock();  
    }
```



Lock
the
access
to
balance



```
    public void withdraw(double amount) {  
        if (amount < 0 || amount > this.balance) {
```



wait(), notify(), and notifyAll()

- Use the **wait()**, **notify()**, and **notifyAll()** methods to facilitate communication among threads.
- They must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.
- The **wait()** method lets the thread wait until some condition occurs. When it occurs, you can use the **notify()** or **notifyAll()** methods to notify the waiting threads to resume normal execution.
- The **notifyAll()** method wakes up all waiting threads, while **notify()** picks up only one thread from a waiting queue.



wait() and notify()

- When **wait()** is invoked, it pauses the thread and simultaneously releases the lock on the object.
- When the thread is restarted **after being notified**, the lock is automatically reacquired.

Example wait(),notify()



```
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try{
                System.out.println("Waiting for b to complete...");
                b.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total;
    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += i;
            }
            notify();
        }
    }
}
```

Output:

Waiting for b to complete...

Total is: 4950



Suspending and resuming threads

- You need two flag variables:
 - One for suspend and resume
 - One for stop

```
boolean suspended;
```

```
boolean stopped;
```

- If `suspended == true` then the thread must pause
- If `suspended == false` then the thread must continue
- If `stopped == true` then the thread must terminate

Example



We implement a thread that counts numbers and it will be suspended, resumed and stopped from the main program

```
public class MyThread implements Runnable{
```

```
    boolean stopped;  
    boolean suspended;
```

} Flags

```
    //constructor
```

```
    public MyThread(){  
        suspended = false;  
        stopped = false;  
    }
```

```
    // the entry point for the thread
```

```
    public void run(){  
        System.out.println("My thread starting.");  
        try{  
            for(int i = 0; i < 1000; i++){  
                System.out.println(i);  
                Thread.sleep(500);
```

} Print numbers

```
                // Use synchronized block to check suspended and stopped  
                synchronized(this){  
                    while(suspended)
```

← If it is suspended then wait...

```
                        wait();  
                    }  
                    if (stopped)
```

← If it is stopped then exit...

```
                        break;
```

```
                }
```

```
    }
```

...continue

```
catch (InterruptedException e) {
    System.out.println("My thread is interrupted");
}
System.out.println("My thread exiting.");
}

//stop the thread
→ synchronized void mystop() {
    stopped = true;
    // the following ensure that a thread that was suspended is stopped
    suspended = false;
    notify();
}

//suspended the thread
→ synchronized void mysuspend() {
    suspended = true;
}

// resume
→ synchronized void myresume() {
    suspended = false;
    notify();
}
}
```



```
public static void main(String[] args) {
    MyThread mythread = new MyThread();

    Thread t = new Thread(mythread);
    t.start();

    try{
        Thread.sleep(1000); // t is running

        mythread.mysuspend(); // suspend the thread
        System.out.println("Suspending thread");
        Thread.sleep(1000);

        mythread.myresume(); // resume the thread
        System.out.println("Resuming thread");
        Thread.sleep(1000);

        mythread.mysuspend(); // suspend again the thread
        System.out.println("Suspending thread");
        Thread.sleep(1000);

        mythread.myresume(); // resume the thread
        System.out.println("Resuming thread");
        Thread.sleep(1000);

        //stopping thread
        mythread.mystop();
    }
    catch (InterruptedException e){
        System.out.println("Main thread interrupted");
    }

    System.out.println("Main thread exiting");
}
```