# Multithreading in Java

- What Are Threads?
- Thread Properties
- Thread Priorities
- Cooperating and Selfish Threads
- Synchronization

## The Concept of Process

A process represents a sequence of **actions** (statements), which are executed **independently** of other actions. For example:

```
process A {
       receive a message;
       print a message;
}
```

**Flow of execution**

```
process B {
       read a number;
       send a number;
}
```

**Flow of execution**

## Parallel and Concurrent Programs

A parallel program consists of a collection of processes executing in parallel. These processes may be executed on one or more physical processors.

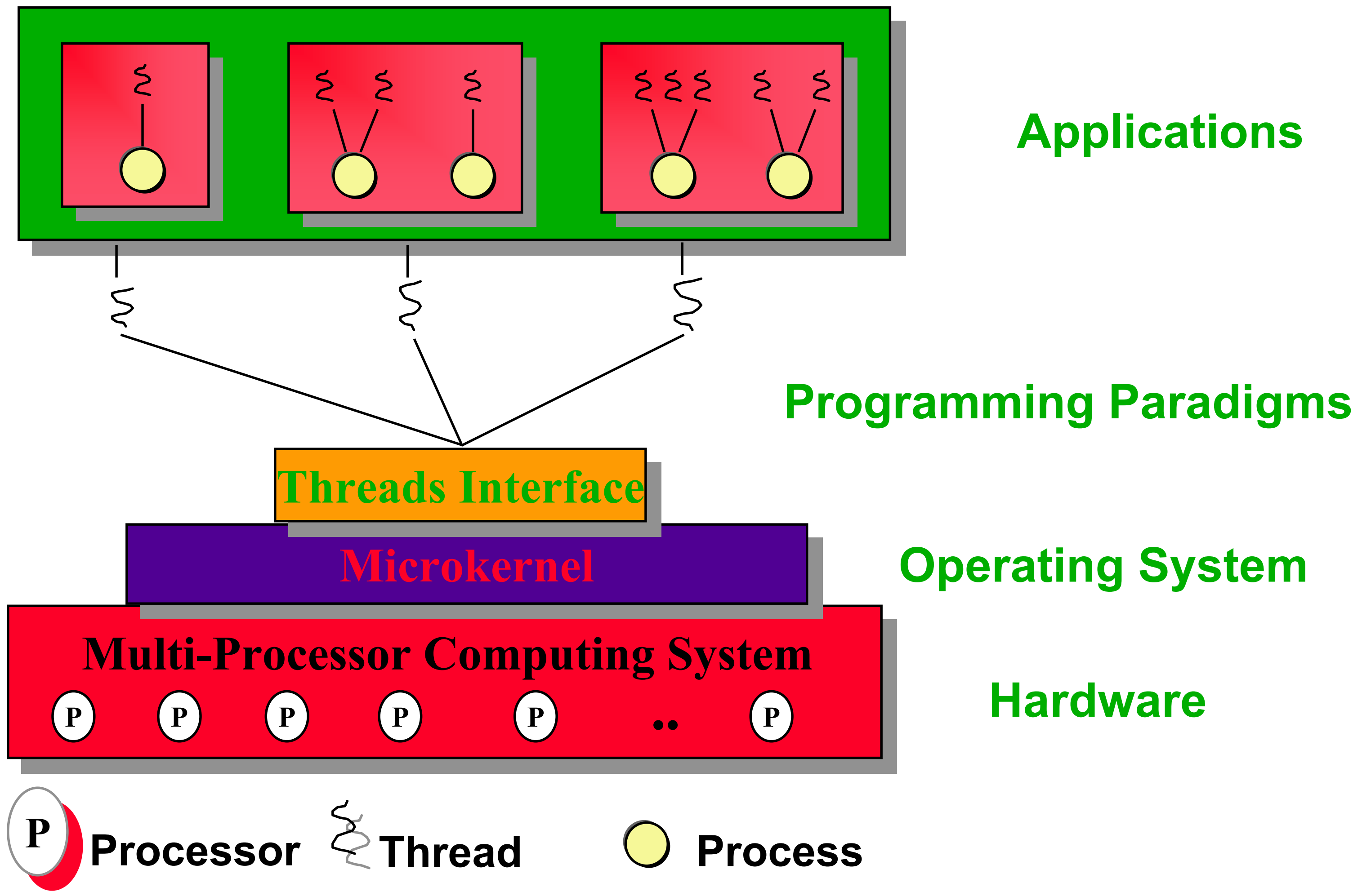
```
 Parallel Composition {
        process A;
        process B;
 }
```

A concurrent program denotes a parallel program where all processes are executed on one physical processor.

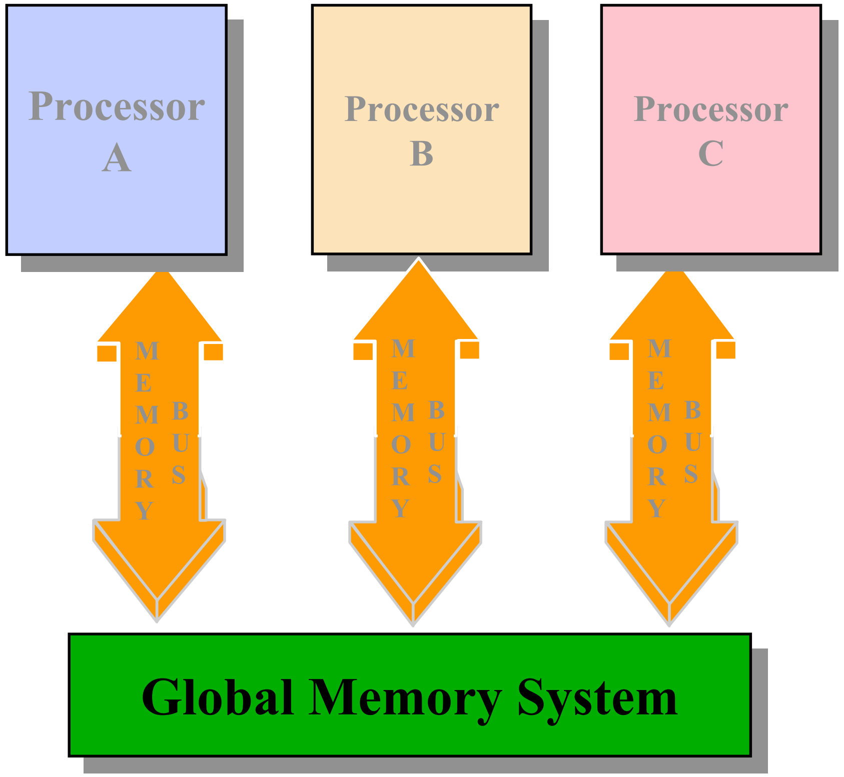
## *Multithreading*

- **Multitasking:**
  - the ability to have **more than one program** **working at the same time** and
  - controlled by the operating system
- **Multithreading**: extends the idea of multitasking by taking it one level lower:
  - **A single program** has the ability to run **multiple computations** **at the same time**
  - Each **computational unit** is called a **thread**.
  - Each thread runs in a different **context**. (Contexts make it seem as though each thread has its own CPU - with registers, memory and code.)
- **Difference** between multiple **processes** and multiple **threads**:
  - Each process has a complete set of its own variables, files, etc.
  - Threads share data in the program in which they live.

# Computing Elements



## Shared Memory MIMD machine



**Comm:** Source PE writes data to GM & destination retrieves it

- ➔ Easy to build, conventional OSes of SISD can be easily be ported
- ➔ **Limitation** : reliability & expandability. A memory component or any processor failure affects the whole system.
- ➔ Increase of processors leads to **memory contention**.

Ex. : SGI PowerChallenge

## *The Bouncing Ball Application*

- The program animates a bouncing ball, finding out if it bounces against a wall, and then redrawing it. As soon as you hit the Start button, the program launches a ball from the upper left corner, and it begins bouncing.
- The handler of the Start button calls the **method** **`bounce()`** of the **class** **`Ball`**, which contains a loop running through 1000 moves.
- After each move, we call the static **`sleep`** method of the **`Thread`** **class** to pause the ball for 5 milliseconds:

```
class Ball
{ . . .
   public void bounce()
   {    draw();
        for (int i = 1; i <= 1000; i++)
        {  move();
        try { Thread.sleep(5); }
        catch(InterruptedException e) {} }
   }
}
```

## *Exercise*

- Study the Bounce.java program. Is the program deterministic?
- Run the Bounce.java program. Is the program deterministic?
- Double the size of the canvas.
- What happens if you remove the `Thread.sleep(5)` delay?
- What happens if you click on the Close button? Is the ball immediately finished bouncing ?

## Using Threads for the Bouncing Ball Application

- We realised that the Close button remains ineffective until the 1000 moves are finished. We cannot interact with the program until the ball has finished bouncing.
- We will make the program more responsive by running the program in **two threads**.
  - One for the bouncing ball
  - another for realising the user interface (**main thread**)
- Since in Java each thread gets a chance to run for a little while, the main thread has the opportunity to notice when you click on the Close button while the ball is bouncing. It can then process the ”close” action.
- **How to create a thread in Java?**
  - Place the code of the thread into the **`run` method of a class derived from `Thread`**.
  - In the example: Derive **`Ball`** from **`Thread`** Rename the **`bounce`** method **`run`**.

```
class Ball extends Thread
{ . . .
   public void run()
   {    draw();
        for (int i = 1; i <= 1000; i++)
        {  move();
        try { sleep(5); }
        catch(InterruptedException e) {} }
   }
}
```

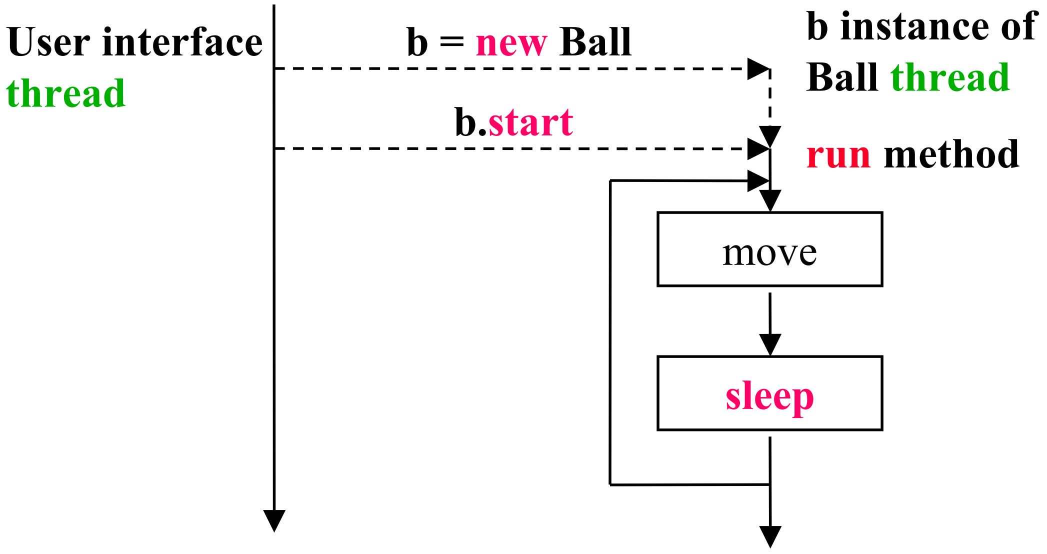
# Running and Starting Threads

- You can construct a thread as an object derived from `Thread`. However, **Java does not automatically starts the thread**.

```
     Ball b = new Ball( . . . );         // won't run yet
```

- **You should call the `start` method in your thread object to start the thread**:

```
     b.start();                          // b will be runnable
```



- If a thread does not need the CPU for a certain time it can tell the other threads explicitly that it is idle. The way to do it is through the `sleep` method.

## Exercise

- Study the BounceThread.java program.
- Compare the code of Bounce.java and BounceThread.java programs.

```
 if (arg.equals("Start"))
         {  Ball b = new Ball(canvas);
            b.bounce(); }           // call function
 if (arg.equals("Start"))
         {  Ball b = new Ball(canvas);
            b.start(); }   // start thread
```

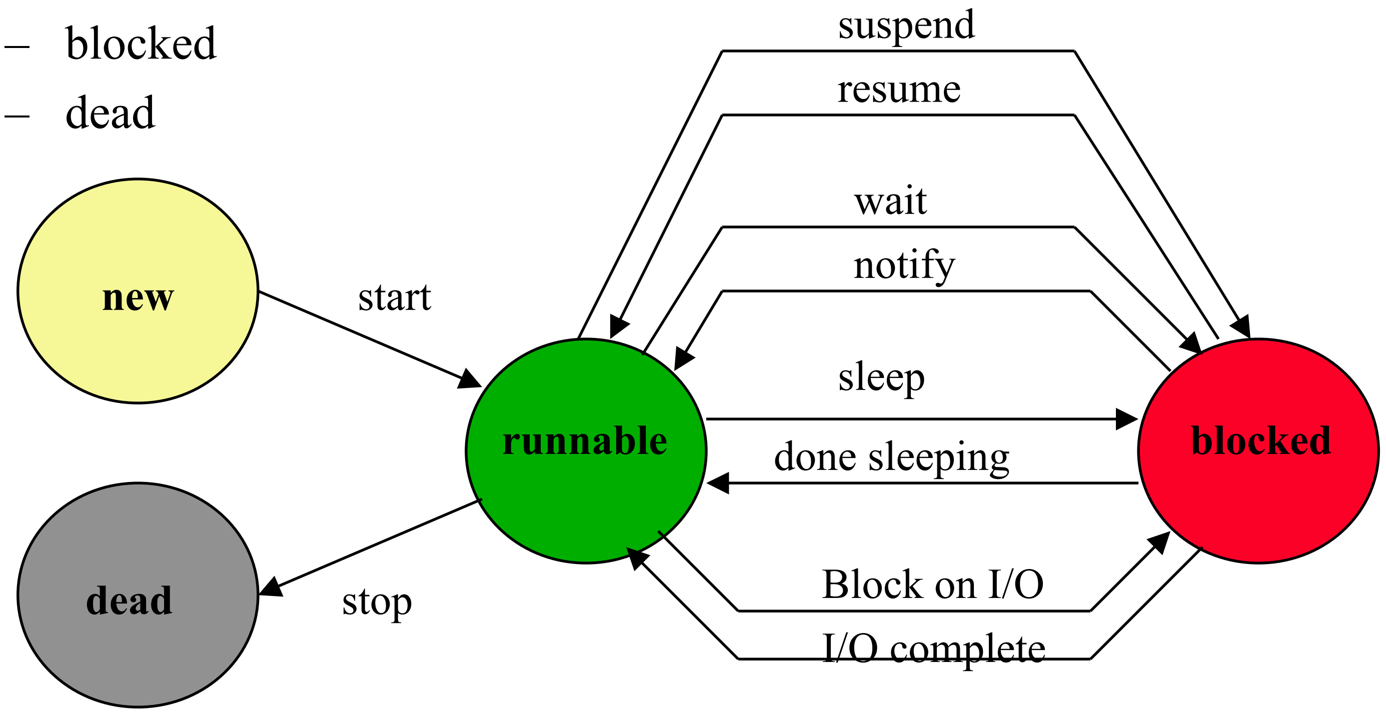
- Run the BounceThread.java program.
- What happens if you click on the Close button?
- Run the BounceThread.java program and click on the Start button several times while a ball is running. What happens? Why?

# Thread States

**Thread states**

- new
- runnable
- blocked
- dead

**State Transition Diagram**



## Thread States (cont.)

- **New**
  - When you create a thread with the **`new` operator**, it is not yet running, it is in the **new state**.
  - The necessary bookkeeping and determining the memory allocation needed are the tasks of the **`start` method**.
- **Runnable**
  - Once you invoke the **`start/run` method**, the thread is **runnable**.
  - A runnable thread **may not yet be running**. It is up to the **operating system** to give it time to run. **Time slicing** is a typical way of running threads.
  - When the code inside the thread begins executing, the thread is **running**.
- **Blocked**

  A thread enters the **blocked state** when one of the four following actions occurs:
  - Someone calls the **`sleep()` method** of the thread.
  - Someone calls the **`suspend()` method** of the thread.
  - The thread calls the **`wait()` method**.
  - The thread calls an operation that is **blocking on I/O**.

# Thread States (cont.)

- **Moving Out of a Blocked State**

  The thread moves out of the blocked state when one of the four following actions occurs:
  - If a thread has been put to **sleep**, the **specified number of milliseconds must expire**.
  - Someone must call its **`resume()`** **method** if the thread was **suspended**.
  - If a thread called the **`wait()`** **method**, then the owner monitor on whose availability the thread is waiting must call **`notify`** or **`notifyAll`**.
  - If a thread is **blocked on I/O**, the **I/O operation must have been finished**.

  If you try to activate a blocked thread in a non-matching way, the thread **remains blocked**.

  If you invoke a method on a thread that is incompatible with its state, the

  **`illegalThreadStateException`**

  is thrown.
- **Dead Threads**

  A thread is dead for one of two reasons:
  - It dies a natural death because **the `run` method exits**.
  - It is **killed** because someone invoked its **`stop`** **method**.
- **Finding out the State of a Thread**

  Use the **`isAlive`** method, which **returns false** if the thread is still new and **not runnable** or if the thread is **dead**.

## Thread Priorities

- Every thread in Java has a priority. By default, **a thread inherits the priority of its parent**.
- You can modify the priority of a thread by the **`setPriority`** method.
- **The range of priority** defined in the `thread` class:
  - MIN_PRIORITY (value = 1)
  - NORM_PRIORITY (value = 5)
  - MAX_PRIORITY (value = 10)
- The **thread scheduler** always starts the **highest-priority runnable thread**.
- A thread keeps running until it either:
  - yields by calling the **`yield`** method,
  - ceases to be runnable (dying or blocked)
  - is replaced by a higher-priority thread that has become runnable.
- **Threads at the same priority** get a turn in **round-robin** fashion.

## Exercise

1. **Study the class `java.lang.Thread`** API description.

   ( file:///C|/java/api/java.lang.Thread.html#_top_ )

2. **Study the code of the BounceExpress.java program which provides a new button called Express. Clicking on the Express button will launch a red ball whose thread runs at a higher priority than the regular balls.**
3. **Show the part of the code responsible for the higher-priority launch of red balls.**
4. **Try it out. Launch one regular ball and many express balls. You will notice that the express balls run faster.**
5. **Explain behaviour 4.**

# Answers

**3. Show the part of the code responsible for the higher-priority launch of red balls.**

```
public boolean action(Event evt, Object arg)
{   if (arg.equals("Start"))
    {   Ball b = new Ball(canvas, Color.black);
        b.setPriority(Thread.NORM_PRIORITY);
        b.start();   }
    }
    else if (arg.equals("Express"))
    {   Ball b = new Ball(canvas, Color.red);
        b.setPriority(Thread.NORM_PRIORITY + 2);
        b.start();
    }
    else if . . .
}
```

**6. Explain it.**

5 milliseconds after an express thread is put to sleep, it is awoken. Then the scheduler again evaluates the priorities of all threads and finds that the express thread has the highest priority, so it gets another turn. If you launch N express balls, they take turns and **only if all are asleep**, do the lower-priority thread gets a chance to run. As the number of express balls increases, the chance that all are asleep decreases.

## Cooperating and Selfish Threads

- A thread should always call **`yield`** or **`sleep`** when it is executing a long loop to ensure that it is **not monopolizing** the system.
  - The **`sleep`** function blocks the thread for a **specified time**
  - The **`yield`** function does not block the thread for any time but it gives a chance to the scheduler to deschedule the thread and dispatch an other one.
- A thread that does not follow this rule and monopolizes the system is called **selfish**.
- The **`BounceSelfish.java`** program shows an example of a selfish thread. When you click on the Selfish button, a blue ball is launched whose run method contains a long loop:

```
public void run()
{   draw();
    for (int i = 1; i <= 1000; i++)
    {     move();
          long t = new Date().getTime();
          while (new Date().getTime() < t + 5)
                  ;
    }
}
```

- The `run` procedure will last about 5 milliseconds before it returns, and never calls `yield` or `sleep`.

## Exercise

1. **Study the class `java.lang.Thread`** API description.

   ( file:///C|/java/api/java.lang.Thread.html#_top_ )

2. **Study the code of the BounceSelfish.java program which provides a new button called Selfish.**
3. **Try it out. Launch a few regular, express, and selfish balls. You will notice that the selfish ball is the fastest and the regular ball is the slowest.**
4. **Explain it.**
5. **What happens if you modify the**

   ```
   while (new Date().getTime() < t + 5)
   ```

   **code into**

   ```
   while (new Date().getTime() < t + 50)
   ```

6. **Explain it.**

## Answers

**4. Explain it.**

The selfish ball keeps the processor between two moves. The only way to take the processor from a selfish ball is by the scheduler when the allocated time-slice is over. The other balls immediately release the processor after a move.

**6. Explain it.**

The selfish ball requires 50 milliseconds to move again. The scheduler applies a time-slice scheduling technique which means each process gets a time-slice to run and then another process can continue for a time-slice. Since the time-slice of the scheduler is much smaller than 50 milliseconds, the other balls can move meanwhile the selfish ball still in the same position.

## Thread Groups

- Some programs have many threads. It then becomes useful to group them by functionality. It is handy to have a way of killing all threads in the same group.
- You **construct a thread group** with the constructor:

```
ThreadGroup g = new ThreadGroup(string)
```

- The string identifies the group and must be unique.
- **To find out if any threads of a group are still runnable**, use the `activeCount` method:

```
if (g.activeCount() == 0)
{   // all threads in the group g have stopped
    . . .   }
```

- **To kill all threads** in a group, call `stop` on the group object:

```
g.stop   //   stops all threads in g
```

- Thread groups can have **child subgroups**. Methods like `activeCount` and `stop` refer to all threads in their group and child groups.
- **Exercise**: **Study the class `java.lang.ThreadGroup`** API description.

  ( file:///C|/java/api/java.lang.ThreadGroup.html#_top_ )

## Thread Communication without Synchronization

- Reading and writing a **shared object** by several threads in parallel without synchronization will lead to serious **non-deterministic errors**.
- An example to illustrate the problem is the **`UnsynchBankTest.java`** application.
- It simulates a bank with **10 accounts**. Randomly generates transactions that move money between these accounts. There are **10 threads**, one for each account.
- **Each transaction** will move a random amount of money from the account serviced by the thread to another random account.
- When this simulation runs we do not know how much money is in any one bank account at any time. But we do know that the **total amount of money** in all the accounts **should remain unchanged** since all we do is move money from one account to another.
- **Exercise 1:** study the code of the **`UnsynchBankTest.java`** program
- **Exercise 2:** run the code and check the total amount of money.
  - Is it always the same?
  - Explain what happens.

## Synchronizing Access to Shared Resources

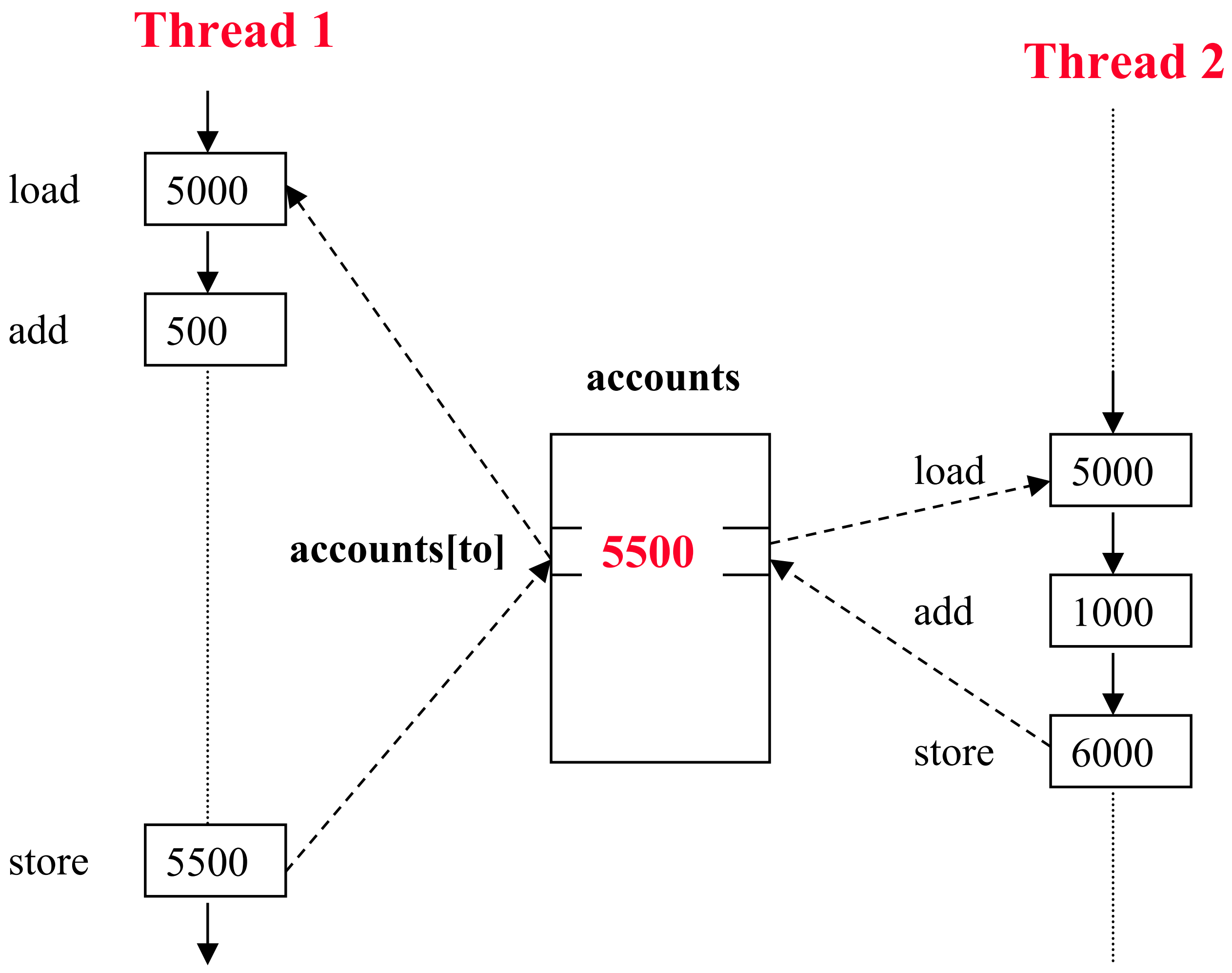
- **Problem** in the previous program can occur when **two threads are simultaneously trying to update** an account:

```
accounts[to] += amount;
```

- The problem is that these are not ***atomic operations***. It represents a series of operations that can be interrupted at any point:
    1. Load `accounts[to]` into a register
    2. Add `amount`
    3. Move result back to `accounts[to]`

- One possible execution order is shown in the next slide. It results the loss of the update of Thread 2.

# Simultaneous Access by Two Thread



## The Principles of Monitors

Although semaphores provide objects to the synchronisation of processes, they are rather "low level" and error-prone. A slight error in the sequence of the synchronisation primitives can lead to serious errors. It is also easy to forget to protect shared variables!

A Monitor is a special type of shared Abstract Data Type (ADT). "*A collection of associated data and procedures is known as a monitor*" (C.A.R Hoare).

## The Principles of a Monitor (Cont.)

"An ADT is a type whose internal form is hidden behind a set of access functions. Objects of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined."

Abstract data types are central to object-oriented programming where every class can be regarded as an ADT.

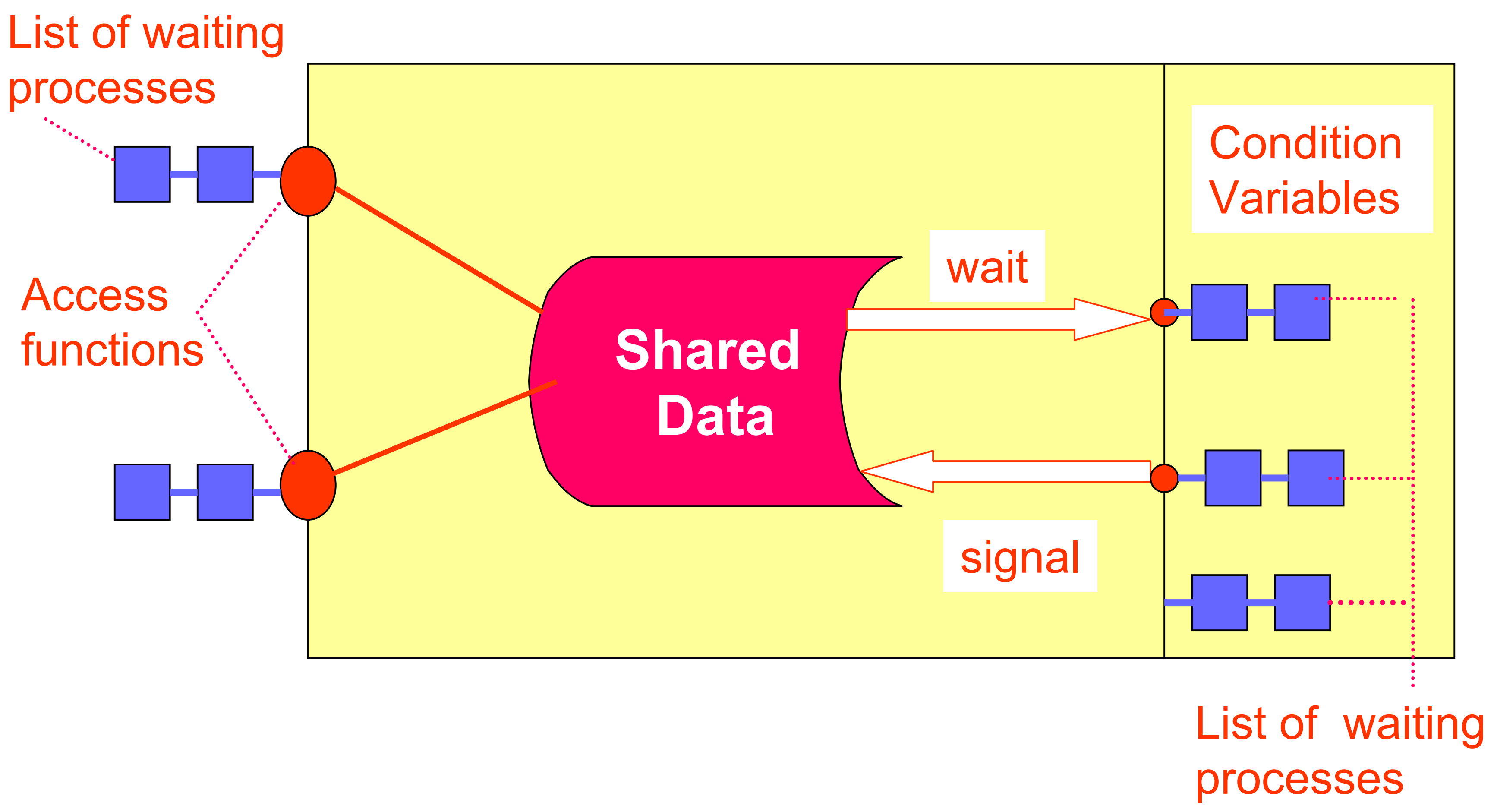
## The Definition of a Monitor

Definition: A monitor is a programming language construct which encapsulates variables, access procedures and initialisation code within an abstract data type.

The monitor's variables (shared data) may only be accessed via its access procedures (methods) and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections.

A monitor may have a queue of processes which are waiting to access it.

## The Structure of a Monitor



## How a Monitor Works

1. Each thread calls an access function. Only one of them is allowed to enter into the monitor at a time. Threads wait in a queue, and when no other process is inside the monitor, one thread of the queue has access to the monitor.

2. Condition variables can be associated with some logical condition on the state of the monitor (some expression that may be either true or false). If a thread discovers (through the access function) that some logical condition it needs is not satisfied, it waits on the corresponding condition variable.

## How a Monitor Works (Cont.)

**3.** Whenever a thread makes one of these conditions true, it signals the corresponding condition variable. When the waiting thread wakes up, it may immediately proceed. Thus, the signaller is blocked on another queue when it calls the signal method and immediately wakes up the waiter (if there are multiple threads blocked on the same condition variable, the one waiting the longest wakes up).

**4.** When a thread leaves the monitor (the access functions terminates), a sleeping signaler, if any, is allowed to continue. Otherwise, the monitor is released, allowing a new thread to enter the monitor. *In summary, waiters usually have precedence over signalers.*

## Implementing Monitors in Java

Java has a kind of built-in monitor, but with two key differences from the general concept of monitor:

First, instead of marking a whole class as monitor, remember to mark each method as **`synchronized`**. Every object is potentially a monitor.

Second, there are no explicit condition variables. In effect, every monitor has exactly one anonymous condition variable. Instead of writing c.wait() or c.signal(), where c is a condition variable, you simply write **`wait()`** inside a loop, which checks the values of the condition, and uses **`notifyAll()`** `to signal.`

## **Example of Monitor:** A Bounded Buffer

```
class BoundedBuffer {
  private Buffer b = new Buffer(10);
  private int count = 0;
synchronized public void put(int item) {
   while (b.isFull()) try{wait();}catch(Exception e){}
         // wait for condition
       b.put(item); // uses the monitor
       notifyAll();  // releases monitor
  }

synchronized public int get() {
    while (b.isEmpty()) try{wait();}catch(Exception e){}
    int result = b.get();
    notifyAll();
    return result;
  }
}
```

## The Main Process

```
ProducerConsumerMainClass {

  public static void main ( String args[] ) {
     BoundedBuffer b = new BoundedBuffer();
     {// Parallel Composition
          new Consumer ( b );
          new Producer ( b );
     }
   }
}
```

**Notice:** The BoundedBuffer object becomes **shared** since it appears in the instance field of both the Consumer and Producer.

## Example of Monitor: The Producer

```
class Producer extends Thread {
    private BoundedBuffer b;
    private int item = 0;
    public Producer ( BoundedBuffer b  ) {
          this.b = b;    // the buffer monitor becomes
                         // part of the instance field
          start();       // starts its own run method
    }
    public void run () {
      for ( int i=0; i< 4; i++ ) { // for test only
       System.out.println("Producer: enter monitor");
       b.put(++item);
       System.out.println("Producer: put =" + item);
       System.out.println("Producer: release" +
                                  "monitor");
         }
    }
}
```

## Example of Monitor: The Consumer

```
class Consumer extends Thread {
    private BoundedBuffer b;
    private int item = 0;
    public Consumer ( BoundedBuffer b  ) {
          this.b = b;    // the buffer monitor becomes
                         // part of the instance field
          start();       // starts its own run method
    }
    public void run () {
      for ( int i=0; i< 4; i++ ) { // for test only
       System.out.println(”Consumer: enter monitor");
       System.out.println(”Consumer: get ="+ b.get());
       System.out.println(”Consumer: release” +
                                 “monitor");
         }
    }
}
```

## Summary of monitors

The key of concepts such as monitor is to provide *a separate object* (ADT), which maintains and controls the access to the shared object.

A monitor is a programming language construct which encapsulates variables, access procedures and initialisation code within an abstract data type.

Java has a built-in monitor, but each method has to be marked as synchronized and there are no explicit condition variables. Simply write **`wait()`** inside a loop, which checks the values of the condition, and use **`notifyAll()`** to signal the waiting processes.

# Returning to the bank problem: Use of Monitors

- The whole `transfer` method can be made atomic by using the **`synchronized`** tag:

```
public synchronized void transfer(int from, int to, int amount)
{   while (accounts[from] < amount)
    {   try { wait(); } catch(InterruptedException e) {}
    }
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % 5000 == 0) test();
}
```

- It realizes **mutual exclusion**: when one thread enters a **`synchronized`** method, Java guarantees that it can finish it before another thread can execute any **`synchronized`** method on the same object. It is the realization of Hoare’s **monitor**. In Java, any object with one or more **`synchronized`** methods is a monitor.
- The second calling thread cannot continue, it is deactivated and placed in a **queue** attached to the monitor object. When the first thread has completed its work with the monitor object, the highest priority thread in the monitor’s waiting queue gets the next turn.

# Synchronization Inside the Monitor

- **Problem**: What do we do when there is not enough money in the account? We wait until some other thread has added funds. But this thread has just gained exclusive access to the bank object, so no other thread has a chance to make a deposit.
- **Solution**: Use the **`wait - notifyAll`** synchronization calls:

```
public synchronized void transfer(int from, int to, int
amount)
 {   while (accounts[from] < amount)
     {   try { wait(); } catch(InterruptedException e) {}
     }
     accounts[from] -= amount;
     accounts[to] += amount;
     ntransacts++;
     if (ntransacts % 5000 == 0) test();
     notifyAll();
 }
```

## Synchronization Inside the Monitor

- **Use of `wait`:**

  When a thread calls `wait` inside a synchronized method, it is deactivated, and Java puts it in the waiting queue of the monitor object.

  (This lets in another thread that can, hopefully, change the account balance.)

- **Use of `notifyAll`:**

  Java awakes a thread that called `wait` when another method calls the `notifyAll` method.

  (This is the signal that the state of the bank object has changed and that waiting threads should be given another chance to inspect the object state. If the ballance is sufficient, it performs, the transfer. If not, it calls `wait` again.)

- **Warning**:

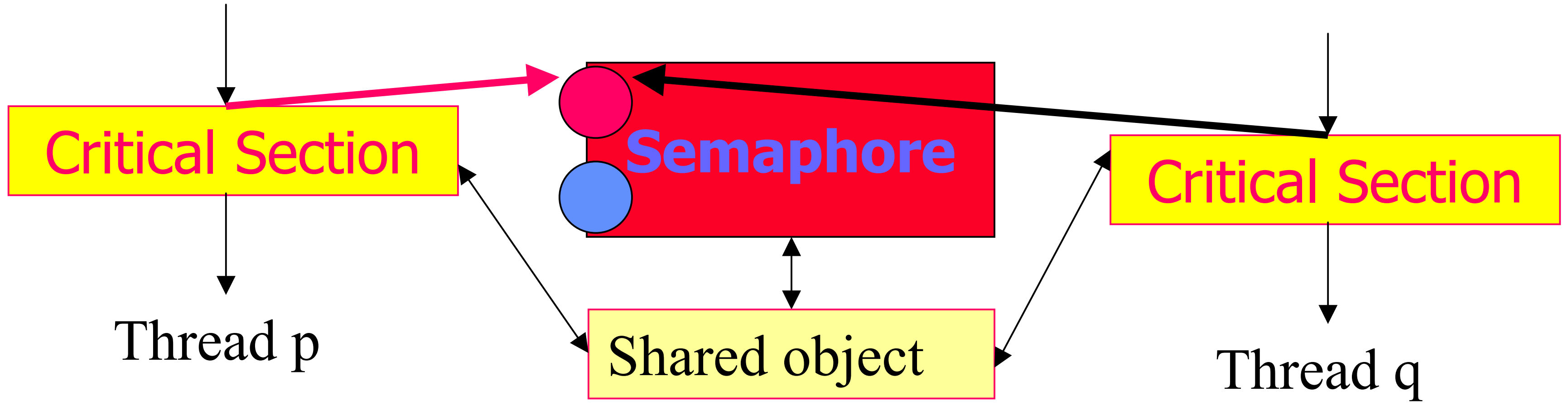
  It is important that the `notifyAll` method is called by some thread - otherwise, the thread that called `wait`, will wait forever. The waiting threads are ***not automatically*** reactivated when no other thread is working on the object.

- **Exercise**: Run the SyncBankTest.java program:
  - Is it correct?
  - What happens if we replace notifyAll with notify?

## The Principle of a Semaphore

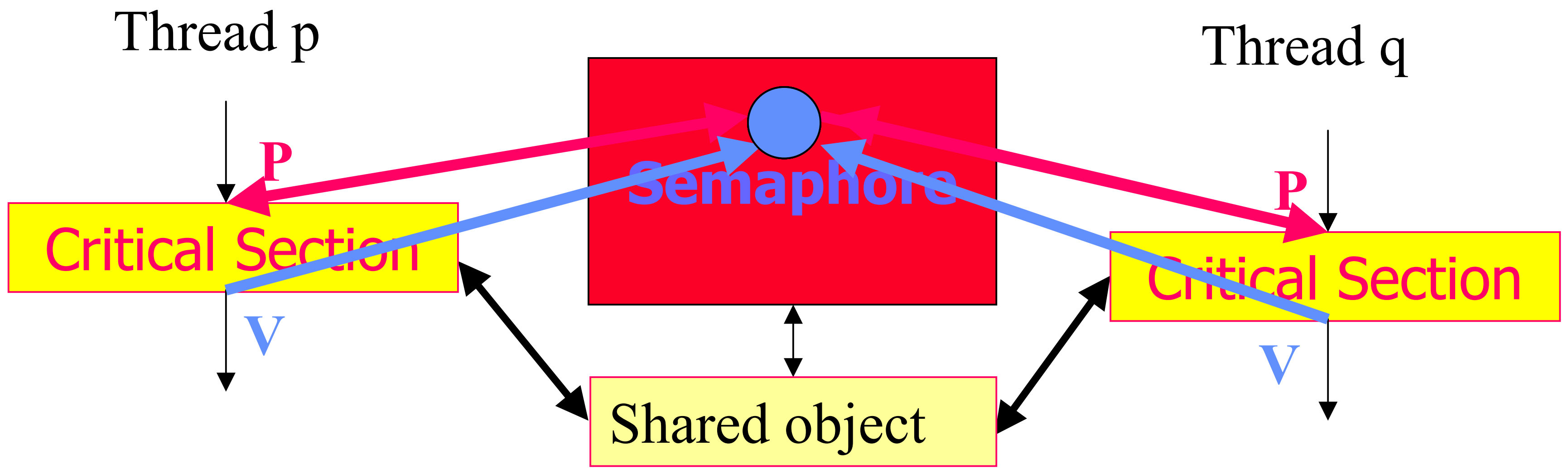
Synchronisation operations require direct contact between the processes. The key of concepts such as semaphore is to provide *a separate object*, where the synchronisation takes place.

The semaphore metaphor is similar to a real semaphore (traffic light). If the light is **red**, the process waits, if it is **green**, it can enter the critical section.



## The Structure of a Semaphore

Definition: A binary semaphore is a mechanism to control access to a critical section based on two (primitive) operations P() and V().



## How a Semaphore Works

**1.** Each process, on entry into the critical section, must call the operation **P()** and on exit from the critical section must call the operation **`V()`**.

**2.** The value is never permitted to be negative. If the value is zero when a process calls **`P()`**, that process is forced to wait until some other process calls **`V()`** on the semaphore.

**3.** The P () and V () operations are atomic. A correct implementation must make it appear that they occur instantaneously. In other words, two operations on the same semaphore attempted at the same time must not be interleaved.

## Providing  Atomicity in Java

Lack of atomicity can lead to race conditions.

A semaphore only works if the operations P() and V() are atomic since the semaphore itself is a shared object.

Methods can become be executed *atomically* in Java by using the keyword **synchronized**. So, if two threads call a synchronized method only one of them has access to the method. The other thread must wait.

```
class Semaphore {
        private int value;
        public Semaphore(int v) { value = v; }
        public synchronized void P() { /* ... */ }
        public synchronized void V() { /* ... */ };
    }
```

## Implementing Semaphores in Java

```
public class Semaphore {
  private int value = 0;
  Semaphore(int n) {
    value = n;
  }
public synchronized void P() {
  while ( value <= 0 ) {
    try {wait();} catch( InterruptedException e ) {}
  }
  value --;
}

public synchronized void V() {
  value ++;
  notifyAll();
  }

}
```

## Example of Semaphore: The Process Behaviour

```
public void run () {
    // Attempts to enter the Critical Section
    s.P();
    // Enters the Critical Section
    data.print();
    data.update(v); // Updates the shared data
    data.print();
    InternalComputation();
    // Reads the final shared data
    data.print();

    s.V();
    //  Leaves the Critical Section
    }
}
```

## The Main Process

```
class ControlledSharedObjectMain{
  public static void main (String args[]) {

    SharedData data = new SharedData();
    Semaphore s = new Semaphore(1);

    ControlledSharedThreadObject t1 = new
      SharedThreadObject("t1", data, 1, s);

    ControlledSharedThreadObject t2 = new
     SharedThreadObject("t2", data, 2, s);

}
```

## Summary

The key of concepts such as semaphore is to provide *a separate object*, where the synchronisation takes place.

A semaphore is based on two (primitive) operations P(), called on entry into the critical section, and **`V()`**, called on exit. If the value is zero when a process calls **`P()`**, that process is forced to wait until some other process calls **`V()`**.

Semaphores are implemented in Java using synchronized methods to provide atomicity and wait and notify to provide synchronisation.

Thank you

## The three ways of starting threads

1. **The parent starts the thread:**

```
class PrimeThread extends Thread {
     public void run() {
         // compute primes...
     }
}
```

To start this thread you need to do the following:

```
PrimeThread p = new PrimeThread();
p.start();
...
```

## The three ways of starting threads

2. **The thread object starts itself by its constructor:**

```
class PrimeThread extends Thread {
      public PrimeThread() {
         start();
      }
      public void run() {
         // compute primes...
      }
 }
```

The thread object immediately starts itself when it is created:

```
PrimeThread p = new PrimeThread();
...
```

## Defining a Thread Extending the `Thread` Class

```
class SimpleThread  extends Thread {

   String msg = null;
   int n = 2;
   SimpleThread ( String msg ) {
        this.msg = msg;
        this.start(); // starts thread immediately
   }
   public void run() {
        for (int i=0; i< n; i++)
          System.out.println ( msg ); // action
   }
}
```

## The three ways of starting threads

3. **Using the Runnable interface :**

```
class Primes implements Runnable {
        public void run() {
                // compute primes...
        }
}
```

To start this thread you need to do the following:

```
Primes p = new Primes();
new Thread(p).start();
...
```

In all cases the **run** method plays the same role as the **main** method in sequential programs.

Meanwhile the main method is called by the Java interpreter, the run method is called by the **start** call. You need as many **start** calls as many threads you want to start.

## Defining a Thread Implementing the `Runnable` Interface

```
class SimpleThread  implements Runnable {

   String msg = null;
   int n = 2;
   SimpleThread ( String msg ) {
        this.msg = msg;
        this.run(); // starts thread immediately
   }
   public void run() {
        for (int i=0; i< n; i++)
          System.out.println ( msg ); // action
   }
}
```

## Creating A Parallel Program Using Threads

```
import SimpleThread;
import SimpleThread1;

class  SimpleThreadMain {
  public static void main ( String args[]) {
        System.out.println("SimpleThreadMain: Start" );

        // Parallel Composition
        {
          SimpleThread p1 = new SimpleThread ("thread p1");

          SimpleThread1 p2 = new SimpleThread1 ("thread p2");
        }

        System.out.println ( "SimpleThreadMain: End " );
    }
}
```

## Observing the Execution of Parallel Processes

```
%java
SimpleThreadMain
SimpleThreadMain:
Start
thread p1
thread p1
thread p2
thread p2
SimpleThreadMain:
End
```

## Combining Atomicity and Synchronisation in Java

Atomicity guarantees safe termination of the operations P() and V() but we still need to provide synchronisation.

The **`notifyAll()`** operation (method) notifies all the threads waiting for an object held by the current thread and wakes them up. Typically, one of the waiting threads will grab the object and proceed.

The operation (method) wait causes the current thread to wait (possibly forever) until another thread notifies it of a condition change. You use wait in conjunction with **`notify()`** or **`notifyAll()`** to coordinate the activities of multiple threads using the same resources.

## **Example of Semaphore:** The Constructor

```
class ControlledSharedThreadObject extends
Thread {
   private SharedData data;
   private String name;
   private int v;
   private Semaphore s;
        ...
   // Simulates the internal computation.
   // Blocks for a random time
   private void InternalComputation() {
      try { sleep((long) Math.random()*100);}
       catch (Exception e ){}
   }
```