

# CS6308- Java Programming

V P Jayachitra

Assistant Professor

Department of Computer Technology

MIT Campus

Anna University

# CONCURRENT PROGRAMMING

- **Concurrency:**

- The idea of two program parts, or even two separate programs, executing *concurrently*
- Concurrency also occurs in all four programming paradigms-imperative, object oriented, functional, and logic.
- save enormous amounts of computing resources, both in space and in speed.

- **Concurrency types**

- The single processor setting
  - **Multithreading** : a program runs on a single processor, but it can dynamically divide into concurrent *threads* of control from time to time.
- The interprocess communication (IPC) setting.
  - **client-server**: a program is viewed as a collection of cooperating processes that run over a network and share data.

# CONCURRENT PROGRAMMING

- **Concurrency:**

- A concurrent program is a program designed to have two or more execution contexts executing in parallel.
- *Parallel program* is a concurrent program in which several execution contexts, or threads, are active *simultaneously*
- To model the parallelism in the real world: Air traffic control,
  - Parallel execution of the program on more than one processor will be much more difficult to achieve
- Concurrency also occurs in all four programming paradigms-imperative, object oriented, functional, and logic.
- save enormous amounts of computing resources, both in space and in speed.
- to provide multiple simultaneous services to the user.

## Multiprogramming

tasks multiplex their executions on a **single processor**, Concept of Context Switching is used. increases CPU utilization.  
More than one process in execution.

## Multitasking

tasks multiplex their executions on a **single processor**. Run multiple processes concurrently through time-sharing  
Concept of Context Switching and Time Sharing is used. increases CPU utilization, it also increases responsiveness.

## Multiprocessing

tasks multiplex their executions on a **multiprocessor** system where there is access to shared memory

## Distributed Processing

tasks multiplex their executions on **several processors** which do not share memory

- **Concurrency types: process vs thread**

- Both process and Thread are independent paths of execution but one process can have multiple Threads.

## 1. The single processor setting

- **Multithreading** : a program runs on a single processor, but it can dynamically divide into concurrent *threads* of control from time to time.
  - **multiple threads are executing in a process at the same time**
  - multiple threads to be created within a process, executing independently but concurrently sharing process resources more easily(heap memory, file descriptors) but each Thread has its own Exception handler and own stack
  - provides concurrency within the context of a single process, i.e a separate flow of control that occurs within a process
- Advantage
  - requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more easily.
  - CPU switches between multiple threads of the same program is simple as it uses wait and notify for example in java
- Disadvantage
  - Without the proper use of locking mechanisms, data inconsistency and dead-lock situations can arise. Thread starvation and resource contention issues arise if many threads are trying to access a shared resource
- Example
  - A web server will utilize multiple threads to simultaneous process requests for data at the same time.
  - An image analysis algorithm will spawn multiple threads at a time and segment an image into quadrants to apply filtering to the image.
  - While typing, multiple threads are used to display your document, asynchronously check the spelling and grammar of your document
  - Multiple threads of execution are used to load content, display animations, play a video, and so on.

## 2. The interprocess communication (IPC) setting.

- **Multitasking /client-server**: a program is viewed as a collection of cooperating processes that run over a network and share data.
  - **multiple processes are executing at the same time**
  - Every process has its own memory space, executable code, and a unique process identifier (PID)
  - provides concurrency between processes.
  - logically concurrent execution of multiple programs ,a separate process for each program that can run in parallel
- Disadvantage
  - CPU switches between multiple programs to complete their execution in real time, duplicate resources and share data as the result of more time-consuming interprocess communication.
  - context switching from one process to another is expensive ; Process are heavyweight and require their own address space
- Example
  - Play MP3 music, edit documents in MS Word, surf the Google **Chrome** .**Firefox** use **multi-threading** for their **tabs**, while **Chrome** use multi-processes .

# History and Definitions

- **Uniprogramming:** *one thread at a time*
  - MS/DOS
- **Multiprogramming:** *more than one thread at a time*
  - MULTIX, UNIX, OS/2, Windows NT/2000/XP
- **Concurrent programming** was used in early operating systems to support parallelism in the underlying hardware and to support multiprogramming and time-sharing.
- A **parallel program** is a concurrent program in which several execution contexts, or threads, are active *simultaneously*
- A **distributed program** is a concurrent program that is designed to be executed simultaneously on a network of autonomous processors that do not share main memory, with each thread running on its own separate processor.

# **Multithreading**

# Outline

- Single thread vs Multi thread
- Thread vs Process
- Runnable vs Thread

# Thread

- What is Thread?
  - A **thread** is an independent path of execution within a program.
- Why Thread?
  - a **thread** is the information needed to serve one individual user or a particular service request.
- Is Java single threaded?
  - Every **thread in Java** is created and controlled by the **java.lang.Thread** class.
  - Every Java application has minimum two threads
    - Main thread
    - Garbage collector thread
  - The Java Virtual Machine allows an application to have multiple threads of execution running concurrently



# Single Thread

- Single task at a time.

```
Class Test{
```

```
    Public static void main(String[] args){
```

```
        Add aobj = new Add(5,3);
```

```
        Sub sobj =new Sub(5,3);
```

```
        //...
```

```
    }
```

Single thread

Task 1

Task 2 will start after completion of Task1

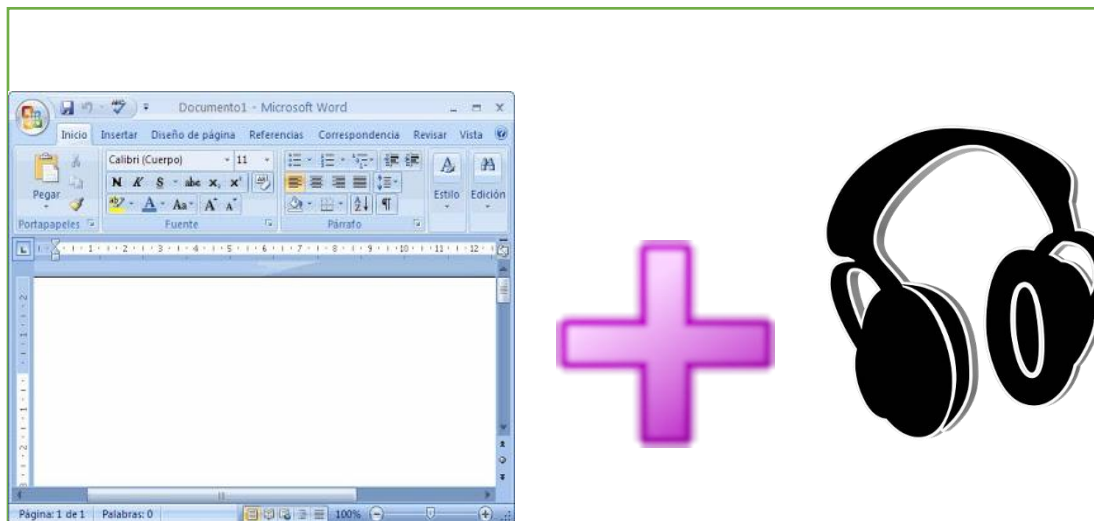
Execution time is more.

# Multithread (multitask)



Multiple task possible for human at a time !

**Do computers perform Multitask at a time?**



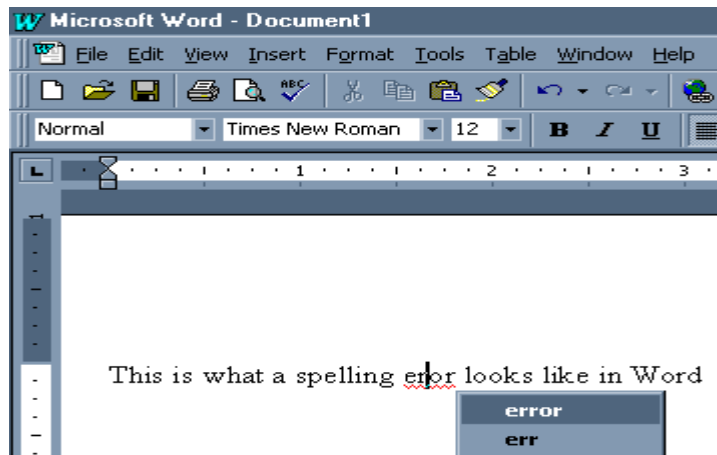
# Use of multithreads

- Reduce execution time
  - Doing tasks in parallel
- Asynchronous tasks
- Web applications-server/client
- Computer games-animations

# Process vs Thread

## Process-heavy weight process

Sequential Program Execution Stream



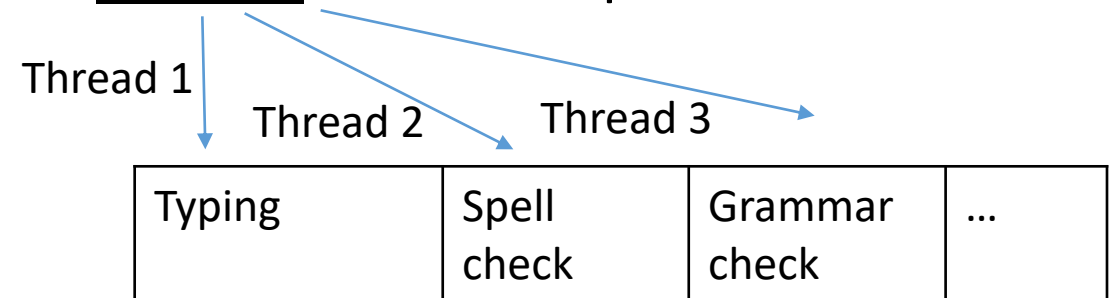
Each application is a process  
Eg. Java application, Ms word , MS  
power point, etc

## Thread –Light weight process

Sub process of a process are threads

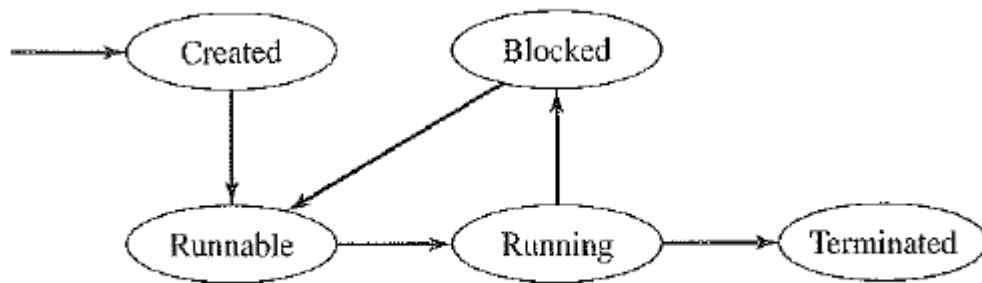
- Type
- Spell check
- Grammar check

Threads are unit of process



# Thread Control and Communication

- 1 **Created**: the thread has been created, but is not yet ready to run.
- 2 **Runnable**: the thread is ready to run (sometimes this state is called *ready*). The thread awaits receiving a processor to run on.
- 3 **Running**: the thread is executing on a processor.
- 4 **Blocked**: the thread is either waiting to enter a section of its code that requires exclusive access to a shared resource (variable), or else has voluntarily given up its processor.
- 5 **Terminated**: the thread has stopped and cannot be restarted.



States of a Thread

Thread transfers back and forth between the *Blocked* and *Running* states :

For example, a thread may be sending several documents to a printer queue, but may need to wait until after the successful printing of one document before it sends a later one.

# Thread Control and Communication

- Concurrent programs require **interthread communication** or interaction.
- **Communication** occurs for the following reasons:
  - 1 A thread sometimes requires exclusive access to a shared resource, like a printer queue, a terminal window, or a record in a data file.
  - 2 A thread sometimes needs to exchange data with another thread.
- In both cases the two communicating threads must synchronize their execution to avoid conflict when acquiring resources, or to make contact when exchanging data.
- A thread can communicate with other threads through:
  - 1 Shared variables: this is the primary mechanism used by Java, and it can also be used by Ada.
  - 2 Message passing: this is the primary mechanism used by Ada.
  - 3 Parameters: this is used by Ada in conjunction with message passing.
- Threads normally cooperate with one another to solve a problem.
- However, it is highly desirable to keep communication between threads to a minimum;

# Races and Deadlocks

- Two fundamental problems that can occur while executing two different threads asynchronously are **race conditions and deadlocks**.
- **Definition:** A **race condition** (sometimes called a *critical race*) occurs when the resulting value of a variable, when two different threads of a program are writing to it, will differ depending on which thread writes to it first.
- The result is determined by which of the threads wins a "race" between them, since it depends on the order in which the individual operations are interleaved over time.

```
c = c + 1;
```

JVM target code

```
1 load c  
2 add 1  
3 store c
```

- In fact, as the number of threads trying to execute this code increases, the resulting number of distinct values computed for *c* can vary between 1 and the number of threads!
- Two thread *a* and *B*, Then the resulting value of *c* depends critically on whether *A* or *B* completes step 3 before the other one begins step 1. If so, then the resulting value of *c* is 2; otherwise the resulting value of *c* is 1.

# Races and Deadlocks

- A thread wishing to acquire a *shared resource*, such as a file or a shared variable (like *c* in the above example), must first acquire access to the resource.
- When the resource is no longer required, the thread must relinquish access to the resource so that other threads can access it.
  - If a thread is unable to acquire a resource, its execution is normally suspended until the resource becomes available.
- Resource acquisition must be administered so that no thread is unduly delayed or denied access to a resource that it needs.
- An occurrence of the latter is often called ***lockout or starvation***.



# Races and Deadlocks

- Errors that occur in a concurrent program may appear as **transient errors**.
- These are errors that may or may not occur, depending on the execution paths of the various threads.
- Finding a transient error can be extremely difficult because the sequence of events that caused the occurrence of the fault may not be known or reproducible.
- Unlike sequential programming, rerunning the same program on the same data may not reproduce the fault.
- Inserting debugging output itself may alter the behavior of the concurrent program so as to prevent the fault from reoccurring.
- Thus, designing a concurrent program is the ability to express it in a form that guarantees the **absence of Critical races**.

# Races and Deadlocks

- The code inside a thread that accesses a shared variable or other resource is termed a *critical section*.
- For a thread to safely execute a critical section, it needs to have access to a locking mechanism; such a mechanism must allow a lock to be tested or set as a single atomic instruction.
- **Locking mechanisms** are used to ensure that only a single thread is executing a critical section (hence accessing a shared variable) at a time;
- This can eliminate critical race conditions and one such locking mechanism is called a semaphore.

# Races and Deadlocks

- The second fundamental problem that can occur while executing two different threads asynchronously is called a **deadlock**.
- **Definition:** A *deadlock* occurs when a thread is waiting for an event that will never happen.
- A deadlock normally involves several threads, each waiting for access to a resource held by another thread.
- A classical example of a deadlock is a traffic jam at an intersection where each car entering is blocked by another
- Four necessary conditions must occur for a deadlock to exist [Coffman *et al.*, 1971]:
  - 1 Threads must claim exclusive rights to resources.
  - 2 Threads must hold some resources while waiting for others; that is, they acquire resources piecemeal rather than all at once.
  - 3 Resources may not be removed from waiting threads (no preemption).
  - 4 A circular chain of threads exists in which each thread holds one or more resources required by the next thread in the chain.
- A thread is said to be **indefinitely postponed** if it is delayed awaiting an event that may never occur: **Unfairness**
- Neglecting fairness in designing a concurrent system may lead to indefinite postponement, thereby rendering the system unusable

# Multithreads: By extending Thread class

Multiple task at a time.

```
Class Test{  
public static void main(String[] args)  
{  
    Add aobj = new Add(5,3);  
    Sub sobj =new Sub(5,3);  
    //...  
    aobj.start();  
    sobj.start();  
}
```

Start method  
Invokes run  
method: creates  
new thread

Class Add **extends** Thread{

```
    int num1;  
    int num2  
    Add(int a, int b){  
        num1 = a; num2 =b;  
    }  
    public void run(){  
        System.out.println("sum="+ num1 +num2);  
    }  
}
```

Subclass extends superclass unless the programmer intends on modifying or enhancing the fundamental behavior of the superclass.

Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.

# Multithreads: By implementing Runnable Interface

Class Test{

public static void main(String[] args)

{

Add aobj = new Add(5,3);

Sub sobj =new Sub(5,3);

//...

Thread t1=new Thread( aobj );

Thread t2=new Thread

t1.start();

t2.start();

}

}

Pass  
runnable ref  
object to  
thread

Start method  
Invokes run  
method:  
creates new  
thread

Class Add implements Runnable{

int num1;

int num2

Add(int a, int b){

num1 = a; num2 =b;

}

public void run(){

System.out.println("sum="+ num1 +num2);

}

}

Runnable interface should be used only if run() method is overridden but not other Thread methods are enhanced.

# Thread constructors

## Constructor Summary

### [Thread\(\)](#)

Allocates a new Thread object.

### [Thread\(Runnable target\)](#)

Allocates a new Thread object.

### [Thread\(Runnable target, String name\)](#)

Allocates a new Thread object.

### [Thread\(String name\)](#)

Allocates a new Thread object.

### [Thread\(ThreadGroup group, Runnable target\)](#)

Allocates a new Thread object.

### [Thread\(ThreadGroup group, Runnable target, String name\)](#)

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

### [Thread\(ThreadGroup group, Runnable target, String name, long stackSize\)](#)

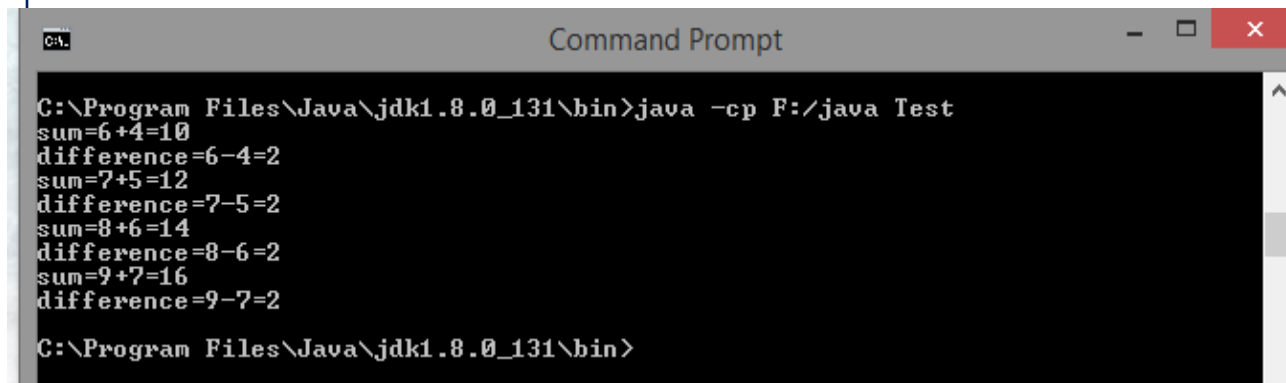
Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified *stack size*.

//multitask using Runnable interface

```
class Add implements Runnable{
    int num1,num2;
    Add(int a, int b){
        num1=a; num2=b;
    }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("sum=" + (num1+i) + "+" +
(num2+i) +"=" + ((num1+i) + (num2+i)));
    }
}
```

```
class Sub implements Runnable{
    int num1,num2;
    Sub(int a, int b){
        num1=a; num2=b;
    }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("difference=" + (num1+i) + "-"
+ (num2+i)+"=" + ((num1+i) - (num2+i)));
    }
}
```

```
class Test{
    public static void main(String[] args){
        Add aobj=new Add(5,3);
        Sub sobj=new Sub(5,3);
        //--->invoke Add and Sub class run() method
        //-->reference object of Add and Sub class is passed
        Thread t1=new Thread(aobj);
        Thread t2=new Thread(sobj);
        t1.start();
        t2.start();
    }
}
```



```
C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java Test
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
difference=8-6=2
sum=9+7=16
difference=9-7=2
C:\Program Files\Java\jdk1.8.0_131\bin>
```

```
//multitask using Thread class
class Add extends Thread{
    int num1,num2;
    Add(int a, int b){
        num1=a; num2=b;
    }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("sum=" + (num1+i) + "+"
+ (num2+i) +"=" + ((num1+i) + (num2+i)));
    }
}

class Sub extends Thread{
    int num1,num2;
    Sub(int a, int b){
        num1=a; num2=b;
    }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("difference=" +
(num1+i) + "-" + (num2+i)+"=" + ((num1+i) - (num2+i)));
    }
}
```

```
class TestDemo{
    public static void main(String[] args){
        Add aobj=new Add(5,3);
        Sub sobj=new Sub(5,3);
        aobj.start();
        sobj.start();
    }
}
```

```
C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestDemo.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestDemo
difference=6-4=2
difference=7-5=2
difference=8-6=2
difference=9-7=2
sum=6+4=10
sum=7+5=12
sum=8+6=14
sum=9+7=16

C:\Program Files\Java\jdk1.8.0_131\bin>
```



```

class Add extends Thread{
    int num1,num2;
    Add(int a, int b){
        num1=a; num2=b; }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("sum=" + (num1+i) + "+"
                               + (num2+i) +"=" + ((num1+i) + (num2+i)));

        try{
            Thread.sleep(1000); }
        catch(Exception e){}
    }
}

class Sub extends Thread{
    int num1,num2;
    Sub(int a, int b){
        num1=a; num2=b; }
    public void run(){
        for(int i=1; i<5; i++)
            System.out.println("difference=" + (num1+i) + "-"
                               + (num2+i)+"=" + ((num1+i) - (num2+i)));

        try{ Thread.sleep(1000); }
        catch(Exception e){}
    } }

```

```

class TestDemo{
    public static void main(String[] args){
        Add aobj=new Add(5,3);
        Sub sobj=new Sub(5,3);
        aobj.start();
        sobj.start();
    }
}

```

```

C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/Test.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java Test
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
difference=8-6=2
sum=9+7=16
difference=9-7=2

C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestDemo.java

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestDemo
sum=6+4=10
difference=6-4=2
sum=7+5=12
difference=7-5=2
sum=8+6=14
sum=9+7=16
difference=8-6=2
difference=9-7=2

```

Write a complete subclass of *Thread* to represent a thread that writes out the numbers from 1 to 10. Then write some code that would create and start a thread belonging to that class.

```
public class CountingThread extends Thread {  
    public static run() {  
        for (int i = 1; i <= 10; i++)  
            System.out.println(i);  
    }  
}
```

```
CountingThread counter; // Declare a variable to represent a thread.  
// Create the thread object.  
counter = new CountingThread();  
// Start the thread running.
```

```
counter.start();
```

```
//create thread object and start thread  
new CountingThread().start();
```

```

class Person implements Runnable{
public static String winner=null;
public void race()
{
    for(int meters=0; meters<=10; meters++){
        if(Person.winner==null){
            if(meters==10){
                Person.winner=Thread.currentThread().getName();
                System.out.println("Winner =" + Person.winner + "meters
run=" + meters);
                break;}
            else
                System.out.println(Thread.currentThread().getName()+
"meters run=" + meters);
        }
    }
}
}

```

```

}
}
public void run()
{ this.race();
}
}

```

```

class TestRace{
public static void main(String[] args){
    Person p1=new Person();
    Person p2=new Person();
    Person p3=new Person();
    Person p4=new Person();
    Thread t1=new Thread(p1,"KAMAL");
    Thread t2=new Thread(p2,"RAJINI");
    Thread t3=new Thread(p3,"AJITH");
    Thread t4=new Thread(p4,"VIJAY");
    t1.start();
    t2.start();
    t3.start();
    t4.start();}}

```

```

C:\Program Files\Java\jdk1.8.0_131\bin>javac F:/java/TestRace.java

```

```

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestRace

```

```

VIJAYmeters run=0
VIJAYmeters run=1
VIJAYmeters run=2
AJITHmeters run=0
AJITHmeters run=1
AJITHmeters run=2
RAJINImeters run=0
RAJINImeters run=1
RAJINImeters run=2
KAMALmeters run=0
KAMALmeters run=1
KAMALmeters run=2
KAMALmeters run=3
RAJINImeters run=3
AJITHmeters run=3
AJITHmeters run=4
AJITHmeters run=5
AJITHmeters run=6
AJITHmeters run=7
VIJAYmeters run=3
AJITHmeters run=8
AJITHmeters run=9
RAJINImeters run=4
KAMALmeters run=4
Winner =AJITHmeters run=10
VIJAYmeters run=4

```

```

KAMALmeters run=4
Winner =AJITHmeters run=10
VIJAYmeters run=4

```

```

C:\Program Files\Java\jdk1.8.0_131\bin>java -cp F:/java TestRace

```

```

RAJINImeters run=0
RAJINImeters run=1
KAMALmeters run=0
KAMALmeters run=1
KAMALmeters run=2
KAMALmeters run=3
KAMALmeters run=4
KAMALmeters run=5
AJITHmeters run=0
VIJAYmeters run=0
VIJAYmeters run=1
VIJAYmeters run=2
VIJAYmeters run=3
AJITHmeters run=1
KAMALmeters run=6
KAMALmeters run=7
KAMALmeters run=8
KAMALmeters run=9
RAJINImeters run=2
Winner =KAMALmeters run=10
AJITHmeters run=2
VIJAYmeters run=4

```

```

class Person implements Runnable{
public static String winner=null;
public synchronized void race()
{
    for(int meters=0; meters<=10; meters++){
        if(Person.winner==null){
            if(meters==10){
                Person.winner=Thread.currentThread().getName();
                System.out.println("Winner =" + Person.winner + "meters
run=" + meters);
                break;}
            else
                System.out.println(Thread.currentThread().getName()+
"meters run=" + meters);
        }
    }
}

public void run()
{ this.race();
}
}

```

```

Elon musk meters run=0
Elon musk meters run=1
Bill gates meters run=0
Elon musk meters run=2
Zuckerberg meters run=0
Bill gates meters run=1
Zuckerberg meters run=1
Zuckerberg meters run=2
Zuckerberg meters run=3
Zuckerberg meters run=4
Zuckerberg meters run=5
Zuckerberg meters run=6
Zuckerberg meters run=7
Jeff Bezos meters run=0
Jeff Bezos meters run=1
Elon musk meters run=3
Elon musk meters run=4
Elon musk meters run=5
Elon musk meters run=6
Elon musk meters run=7
Jeff Bezos meters run=2
Jeff Bezos meters run=3
Zuckerberg meters run=8
Bill gates meters run=2
Zuckerberg meters run=9
Jeff Bezos meters run=4
Elon musk meters run=8
Bill gates meters run=3
Winner =Zuckerbergmeters
run=10

```

```

class TestRace{
public static void main(String[] args){
    Person p1=new Person();
    Person p2=new Person();
    Person p3=new Person();
    Person p4=new Person();
    Thread t1=new Thread(p1,"Elon musk");
    Thread t2=new Thread(p2,"Jeff Bezos");
    Thread t3=new Thread(p3,"Zuckerberg");
    Thread t4=new Thread(p4,"Bill gates");
    t1.start();
    t2.start();
    t3.start();
    t4.start();}}

```

```

Elon musk meters run=0
Elon musk meters run=1
Elon musk meters run=2
Elon musk meters run=3
Elon musk meters run=4
Elon musk meters run=5
Elon musk meters run=6
Elon musk meters run=7
Elon musk meters run=8
Elon musk meters run=9
Zuckerberg meters run=0
Bill gates meters run=0
Jeff Bezos meters run=0
Winner =Elon muskmeters run=10

```

# SYNCHRONIZATION STRATEGIES

- Two principal devices have been developed that support programming for concurrency:
  - semaphores and monitors.

# SYNCHRONIZATION STRATEGIES: Semaphores

- Semaphores were originally defined by Dijkstra [1968a].
- Basically, a semaphore is an integer variable and an associated thread queueing mechanism.
- Two atomic operations, traditionally called  $P$  and  $V$ , are defined for a semaphores:
  - $P(s)$ - >if  $s > 0$  then assign  $s = s - 1$ ; otherwise block (enqueue) the thread that called  $P$ .
  - $V(s)$ - >if a thread  $T$  is blocked on the semaphore  $s$ , then wake up  $T$ ; otherwise assign  $s = s + 1$ .
- The operations  $P$  and  $V$  are atomic in the sense that they cannot be interrupted once they are initiated.
  - If the semaphore only takes on the values 0 and 1, it is called a *binary* semaphore. Otherwise, it is called a *counting semaphore*.

# Semaphores : producer-consumer

- A classic example occurs in the case of producer-consumer cooperation, also called as *cooperative synchronization*.
- Where a **single producer task deposits information** into a shared, single-entry buffer for a **single consumer task to retrieve**.
- The **producer** thread waits (via a P) for the buffer to be empty, deposits information, then signals (via a V) that the buffer is full.
- The **consumer** thread waits (via a P) for the buffer to be full, then removes the information from the buffer, and signals (via a V) that the buffer is empty.
- semaphore is an elegant, low-level mechanism for synchronization control, difficult to build a large, multitasking system like OS.

# Monitors

- *Monitors* [Hoare, 1974] provide an alternative device for managing concurrency and avoiding deadlock.
- Monitors provide the basis for synchronization in Java.
  - Provide an automatic locking mechanism on concurrent operations so that at most one thread can be executing an operation at one time.
  - Its purpose is to encapsulate a shared variable with primitive operations (signal and wait) on that variable



```

public class RaceCondition{
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(counter, "Thread-1");
        Thread t2 = new Thread(counter, "Thread-2");
        Thread t3 = new Thread(counter, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}

```

### Output

```

Value After increment Thread-2 3
Value after decrementing Thread-2 2
Value After increment Thread-1 2
Value after decrementing Thread-1 1
Value After increment Thread-3 1
Value after decrementing Thread-3 0

```

```

class Counter implements Runnable{
    private int c = 0;
    public void increment() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) { e.printStackTrace(); }
        c++;
    }
    public void decrement() {
        c--; }

    public int getValue() {
        return c; }

    @Override
    public void run() {
        this.increment(); //incrementing
        System.out.println("Value After increment "
            + Thread.currentThread().getName() + " " + this.getValue());
        this.decrement(); //decrementing
        System.out.println("Value after decrementing "
            + Thread.currentThread().getName() + " " + this.getValue());
    }
}

```

```

public class RaceCondition{
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(counter, "Thread-1");
        Thread t2 = new Thread(counter, "Thread-2");
        Thread t3 = new Thread(counter, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}

```

## Output

```

Value After increment Thread-2 1
Value after decrementing Thread-2 0
Value After increment Thread-3 1
Value after decrementing Thread-3 0
Value After increment Thread-1 1
Value after decrementing Thread-1 0

```

```

class Counter implements Runnable{
    private int c = 0;
    public void increment() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) { e.printStackTrace(); }
        c++;
    }
    public void decrement() {
        c--; }

    public int getValue() {
        return c; }

    @Override
    public void run() {
        synchronized(this){
            this.increment(); //incrementing
            System.out.println("Value After increment "
                + Thread.currentThread().getName() + " " + this.getValue());
            this.decrement(); //decrementing
            System.out.println("Value after decrementing "
                + Thread.currentThread().getName() + " " + this.getValue());
        } } }

```

# java.util.concurrent

- Defines the built-in approaches to **synchronization and interthread communication**.
- These include
  - Synchronizers
  - Executors
  - Concurrent collections
  - The Fork/Join Framework

# Synchronizers

- Synchronizers offer high-level ways of synchronizing the interactions between multiple threads.
- The synchronizer classes defined by `java.util.concurrent` are

Class	Explanation
Semaphore	Implements the classic semaphore.
CountDownLatch	Waits until a specified number of events have occurred.
CyclicBarrier	Enables a group of thread to wait at a predefined execution point.
Exchanger	Exchange data between threads.
Phaser	Synchronized threads that advance through multiple phases of an operation.

# Semaphore

- To access the resource, a thread must be granted a permit from the semaphore.
- A semaphore controls access to a shared resource through the use of a counter.
  - If the counter is greater than zero, then access is allowed. If it is zero, then access is denied.

# Semaphore

- **To use a semaphore:**

- Thread that wants access to the shared resource tries to acquire a permit.
  - If the semaphore's count is greater than zero, then the thread acquires a permit
  - Then the semaphore's count is decremented.
  - Otherwise, the thread will be blocked until a permit can be acquired.
- Thread no longer needs access to the shared resource, it releases the permit
  - Then the semaphore's count is incremented.
  - If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

# Java's Semaphore class

- Semaphore has the two constructors :

```
Semaphore(int num)  
Semaphore(int num, boolean how)
```

- **num** specifies the initial permit count.
  - i.e., num specifies the number of threads that can access a shared resource at any one time.
- If **num is one**, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order.
- Setting **how to true** ensure that waiting threads are granted a permit in the order in which they requested access.

# Java's Semaphore class

- To acquire a permit, call the `acquire( )` method, which has these two forms:

```
void acquire( ) throws InterruptedException  
void acquire(int num) throws InterruptedException
```

- The first form acquires one permit.
- The second form acquires `num` permits.
- If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.



# Java's Semaphore class

- To release a permit, call `release( )`, which has these two forms:

```
void release( )  
void release(int num)
```

- The first form releases one permit.
- The second form releases the number of permits specified by `num`.
- To use a semaphore to control access to a resource, each thread that wants to use that resource must first call `acquire( )` before accessing the resource.
- When the thread is done with the resource, it must call `release( )`.

```

import java.util.concurrent.Semaphore;

class IncThread implements Runnable{
    String name;
    Semaphore sem;
    IncThread(Semaphore sem, String name){
        this.name =name;
        this.sem=sem;    }
    public void run(){
        System.out.println(name+": "+"is starting");
        System.out.println(name+": "+"waiting for permit");
        try{ //first acquire permit
            sem.acquire();
            System.out.println(name+": "+"acquired permit");
            //now access shared resource
            for(int i=1; i<=5; i++) {
                Shared.count++;
                System.out.println(name+": "+Shared.count);
                Thread.sleep( time: 10);
            }
            //release permit
            sem.release();
            System.out.println(name+": "+"released permit");
        }
        catch (InterruptedException e1){
            System.out.println(e1);    }
    }
}

class DecThread implements Runnable{
    String name;
    Semaphore sem;
    DecThread(Semaphore sem, String name){
        this.name =name;
        this.sem=sem;
    }
    public void run(){
        System.out.println(name+": "+"is starting");
        System.out.println(name+": "+"waiting for permit");
        try{
            sem.acquire();
            System.out.println(name+": "+"acquired permit");
            for(int i=1; i<=5; i++){
                Shared.count--;
                System.out.println(name+": "+Shared.count);
                Thread.sleep( time: 10);    }
            sem.release();
            System.out.println(name+": "+"released permit");
        }
        catch (InterruptedException e){
            System.out.println(e);    }
    }
}

//A shared resource
class Shared{
    static int count=1; }

public class Synch {
    public static void main(String args[]){
        Semaphore sem=new Semaphore( permits: 1);
        IncThread inc=new IncThread(sem, name: "A");
        DecThread dec=new DecThread(sem, name: "B");
        Thread t1=new Thread(inc);Thread t2=new Thread(dec);
        t1.start();t2.start();
    }
}

```

## Output

```

B:is starting
A:is starting
A:waiting for permit
B:waiting for permit
A:acquired permit
A:2
A:3
A:4
A:5
A:6
B:acquired permit
A:released permit
B:5
B:4
B:3
B:2
B:1
B:released permit

```

# Java's Semaphore class

- In both IncThread and DecThread, notice the call to sleep( ) within run( ).
- It is used to “prove” that accesses to Shared.count are synchronized by the semaphore.
- In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run.
- However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete.
- Thus, Shared.count is incremented five times by IncThread and decremented five times by DecThread.
- The increments and decrements are not intermixed

```
// An implementation of a producer and consumer
// that use semaphores to control synchronization.
```

```
import java.util.concurrent.Semaphore;
```

```
class Q {
    int n;
```

```
    // Start with consumer semaphore unavailable.
```

```
    static Semaphore semCon = new Semaphore(0);
```

```
    static Semaphore semProd = new Semaphore(1);
```

```
    void get() {
```

```
        try {
```

```
            semCon.acquire();
```

```
        } catch (InterruptedException e) {
```

```
            System.out.println("InterruptedException caught");
```

```
        }
```

```
        System.out.println("Got: " + n);
```

```
        semProd.release();
```

```
    }
```

```
    void put(int n) {
```

```
        try {
```

```
            semProd.acquire();
```

```
        } catch (InterruptedException e) {
```

```
            System.out.println("InterruptedException caught");
```

```
        }
```

```
        this.n = n;
```

```
        System.out.println("Put: " + n);
```

```
        semCon.release();
```

```
    }
```

```
}
```

```
class Producer implements Runnable {
```

```
    Q q;
```

```
    Producer(Q q) {
```

```
        this.q = q;
```

```
    }
```

```
    public void run() {
```

```
        for(int i=0; i < 20; i++) q.put(i);
```

```
    }
```

```
class Consumer implements Runnable {
```

```
    Q q;
```

```
    Consumer(Q q) {
```

```
        this.q = q;
```

```
    }
```

```
    public void run() {
```

```
        for(int i=0; i < 20; i++) q.get();
```

```
    }
```

```
}
```

```
class ProdCon {
```

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

```
        Q q = new Q();
```

```
        new Thread(new Consumer(q), "Consumer").start();
```

```
        new Thread(new Producer(q), "Producer").start();
```

```
    }
```

```
}
```

A portion of the output is shown here:

Put: 0

Got: 0

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

.

# Java's Semaphore class

- The sequencing of `put( )` and `get( )` calls is handled by two semaphores: `semProd` and `semCon`.
- Before `put( )` can produce a value, it must acquire a permit from `semProd`.
- After it has set the value, it releases `semCon`.
- Before `get( )` can consume a value, it must acquire a permit from `semCon`.
- After it consumes the value, it releases `semProd`.
- This “give and take” mechanism ensures that each call to `put( )` must be followed by a call to `get( )`.
- Notice that `semCon` is initialized with no available permits. This ensures that `put( )` executes first.
- The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

# CountDownLatch

- Sometimes a thread need to wait until one or more events have occurred.
- A CountDownLatch:
  - A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released.
  - Each time an event happens, the count is decremented.
  - When the count reaches zero, the latch opens.

- **CountDownLatch** has the following constructor:

**CountDownLatch(int num)**

- Here, num specifies the number of events that must occur in order for the latch to open.
- To wait on the latch, a thread calls `await( )`, which has the forms shown here:

**void await( ) throws InterruptedException**

**boolean await(long wait, TimeUnit tu) throws InterruptedException**

- The first form waits until the count associated with the invoking CountDownLatch reaches zero.
- The second form waits only for the period of time specified by wait.
- It returns false if the time limit is reached and true if the countdown reaches zero.
- To signal an event, call the `countDown( )` method, shown next:

**void countDown( )**

- Each call to `countDown( )` decrements the count associated with the invoking object.

The following program demonstrates CountdownLatch. It creates a latch that requires five events to occur before it opens.

```
// An example of CountdownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);

        System.out.println("Starting");

        new Thread(new MyThread(cdl)).start();

        try {
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        System.out.println("Done");
    }
}
```

```
class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
        latch = c;
    }

    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i);
            latch.countDown(); // decrement count
        }
    }
}
```

The output produced by the program is shown here:

```
Starting
0
1
2
3
4
Done
```

- Inside main( ), a CountdownLatch called cdl is created with an initial count of five.
- Next, an instance of MyThread is created, which begins execution of a new thread.
- cdl is passed as a parameter to MyThread's constructor and stored in the latch instance variable.
- Then, the main thread calls await( ) on cdl, which causes execution of the main thread to pause until cdl's count has been decremented five times.
- Inside the run( ) method of MyThread, a loop is created that iterates five times.
- With each iteration, the countDown( ) method is called on latch, which refers to cdl in main( ).
- After the fifth iteration, the latch opens, which allows the main thread to resume.



# The Main Thread

- When a Java program starts up, one thread begins running immediately and is called the main thread of the program.
  - The main thread is important for two reasons:
    - It is the thread from which other “child” threads will be spawned.
    - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Java program can be controlled through a Thread object.
  - To control the main thread, obtain a reference to it by calling the method `currentThread( )`, which is a public static member of Thread.

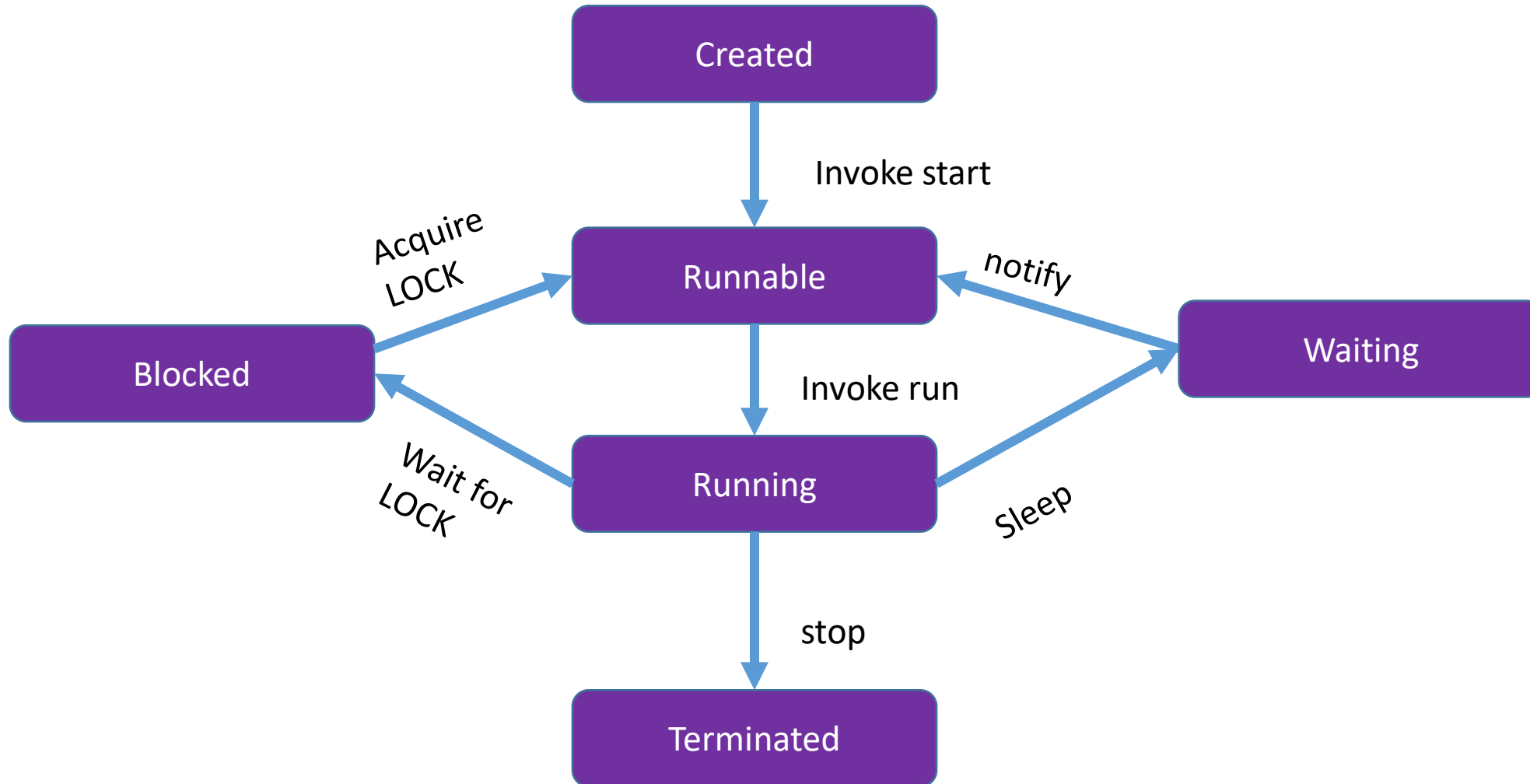
```
static Thread currentThread( )
```

```
public class Thread1 {  
    public static void main(String args[]){  
        Thread t=Thread.currentThread();  
        System.out.println("Name:"+t.getName());  
        System.out.println("state:"+t.getState());  
        System.out.println("Priority:"+t.getPriority());  
        System.out.println("Id:"+t.getId());  
        System.out.println("ThreadGroup:"+t.getThreadGroup());  
        t.setName("My thread");  
        System.out.println("Name:"+t.getName());  
        try {  
            for (int j = 0; j < 5; j++) {  
                Thread.sleep(1000);  
            }  
        }  
        catch (InterruptedException e){  
            System.out.println(e);  
        }  
    }  
}
```

## Output:

```
Name:main  
state:RUNNABLE  
Priority:5  
Id:1  
ThreadGroup:java.lang.ThreadGroup[name=main,maxpri=10]  
Name:My thread
```

# Thread control and communication



# Creating Multiple Threads

- How can one thread know when another thread has ended?
- Two ways exist to determine whether a thread has finished.
  - Using `isAlive( )` and `join( )`
    - call `isAlive( )` on the thread.

**`final boolean isAlive( )`**

- The `isAlive( )` method returns true if the thread upon which it is called is still running. It returns false otherwise.
- Call `join( )` to wait for a thread to finish.

```

    public void run() {
        try {
            for (int j = 0; j < 2; j++) {
                System.out.println("Name:" + Thread.currentThread().getName() + "State:" + Thread.currentThread().getState());
                Thread.sleep( time: 1000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println("Name:" + Thread.currentThread().getName() + "State:" + Thread.currentThread().getState());
    }
}

```

```

public class MainThread {
    public static void main(String args[]){
        Thread t1=new Thread(new Threads(), threadName: "A");
        Thread t2=new Thread(new Threads(), threadName: "B");
        Thread t3=new Thread(new Threads(), threadName: "C");
        t1.start();
        t2.start();
        t3.start();
        System.out.println(t1.getName()+"is alive?" +t1.isAlive());
        System.out.println(t2.getName()+"is alive?" +t2.isAlive());
        System.out.println(t3.getName()+"is alive?" +t3.isAlive());
        try {
            if (t1.isAlive()) t1.join();
            if (t2.isAlive()) t2.join();
            if (t3.isAlive()) t3.join();
        }
        catch(InterruptedException e){
            System.out.println(e);
        }
        System.out.println(t1.getName()+"is alive?" +t1.isAlive());
        System.out.println(t2.getName()+"is alive?" +t2.isAlive());
        System.out.println(t3.getName()+"is alive?" +t3.isAlive());
        if (!(t1.isAlive() && t2.isAlive() && t3.isAlive()))
            System.out.println(Thread.currentThread().getName()+" exiting");
    }
}

```

## Output:

```

Name:CState:RUNNABLE
Ais alive?true
Name:AState:RUNNABLE
Name:BState:RUNNABLE
Bis alive?true
Cis alive?true
Name:BState:RUNNABLE
Name:AState:RUNNABLE
Name:CState:RUNNABLE
Name:AState:RUNNABLE
Name:CState:RUNNABLE
Name:BState:RUNNABLE
Ais alive?false
Bis alive?false
Cis alive?false
main exiting

```

```

    public void run() {
        try {
            for (int j = 0; j < 2; j++) {
                System.out.println("Name:" + Thread.currentThread().getName() + "State:" + Thread.currentThread().getState());
                Thread.sleep( time: 1000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println("Name:" + Thread.currentThread().getName() + "State:" + Thread.currentThread().getState());
    }
}

```

```

public class MainThread {
    public static void main(String args[]){
        Thread t1=new Thread(new Threads(), threadName: "A");
        Thread t2=new Thread(new Threads(), threadName: "B");
        Thread t3=new Thread(new Threads(), threadName: "C");
        t1.start();
        t2.start();
        t3.start();
        System.out.println(t1.getName()+"is alive?" +t1.isAlive());
        System.out.println(t2.getName()+"is alive?" +t2.isAlive());
        System.out.println(t3.getName()+"is alive?" +t3.isAlive());
        /*try {
            if (t1.isAlive()) t1.join();
            if (t2.isAlive()) t2.join();
            if (t3.isAlive()) t3.join();
        }
        catch(InterruptedException e){
            System.out.println(e);
        }
        if (!(t1.isAlive() && t2.isAlive() && t3.isAlive()))*/
        System.out.println(Thread.currentThread().getName()+" exiting");
    }
}

```

## Output:

```

Ais alive?true
Name:CState:RUNNABLE
Name:BState:RUNNABLE
Name:AState:RUNNABLE
Bis alive?true
Cis alive?true
main exiting
Name:AState:RUNNABLE
Name:CState:RUNNABLE
Name:BState:RUNNABLE
Name:AState:RUNNABLE
Name:BState:RUNNABLE
Name:CState:RUNNABLE

```

# Thread priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads.
- To set a thread's priority, use the `setPriority( )` method, which is a member of `Thread`.

**`final void setPriority(int level)`**

- Here, `level` specifies the new priority setting for the calling thread.
- The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`. Currently, these values are 1 and 10, respectively.
- To set a thread to default priority, use `NORM_PRIORITY`, which is currently 5. These priorities are defined as static final variables within `Thread`.

# Thread priorities

- To obtain the current priority use `getPriority( )` method of `Thread`.

```
final int getPriority( )
```



```

public class MainThread {
    public static void main(String args[]){
        Thread t1=new Thread(new Threads(),"A");
        Thread t2=new Thread(new Threads(),"B");
        Thread t3=new Thread(new Threads(),"C");
        Thread t4=new Thread(new Threads(),"D");
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.NORM_PRIORITY);
        t3.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        System.out.println(t1.getName()+" is alive? "+t1.isAlive()+ " With priority " + t1.getPriority());
        System.out.println(t2.getName()+" is alive? "+t2.isAlive()+ " With priority " + t2.getPriority());
        System.out.println(t3.getName()+" is alive? "+t3.isAlive()+ " With priority " + t3.getPriority());
        System.out.println(t4.getName()+" is alive? "+t4.isAlive()+ " With priority " + t4.getPriority());
        try {
            if (t1.isAlive()) t1.join();
            if (t2.isAlive()) t2.join();
            if (t3.isAlive()) t3.join();
        }
        catch(InterruptedException e){ System.out.println(e); }
        System.out.println(t1.getName()+" is alive? "+t1.isAlive()+" With id:" + t1.getId());
        System.out.println(t2.getName()+" is alive? "+t2.isAlive()+" With id:" + t2.getId());
        System.out.println(t3.getName()+" is alive? "+t3.isAlive()+" With id:" + t3.getId());
        System.out.println(t4.getName()+" is alive? "+t4.isAlive()+ " With id:" + t4.getId());
        if (!(t1.isAlive() && t2.isAlive() && t3.isAlive()))
            System.out.println(Thread.currentThread().getName()+" exiting ");
    }
}

```

Output:

```

Name:C State:RUNNABLE
Name:D State:RUNNABLE
A is alive? true With priority 1
Name:B State:RUNNABLE
B is alive? true With priority 5
C is alive? true With priority 10
D is alive? true With priority 5
Name:A State:RUNNABLE
Name:C State:RUNNABLE
Name:D State:RUNNABLE
Name:B State:RUNNABLE
Name:A State:RUNNABLE
Name:C State:RUNNABLE
Name:B State:RUNNABLE
Name:D State:RUNNABLE
Name:A State:RUNNABLE
A is alive? false With id:17
B is alive? false With id:18
C is alive? false With id:19
D is alive? false With id:20
main exiting

```

# Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the **monitor**.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

# Why synchronization?

- All three threads calling the same method, on the same object, at the same time is known as a race condition.
- Here the three threads are racing each other to complete the method.
- In most situations, a race condition is more subtle and less predictable, as it is not sure when the context switch will occur.
- This can cause a program to run right one time and wrong the next.
- Solution: To fix the racing condition, it is necessary to serialize the access to call( ). That is, restrict its access to only one thread at a time.

# Synchronization

- Using Synchronized Methods

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

- Using synchronized Statement or block
  - Further, a third party class methods have not access to the source code.
  - Thus, adding synchronized to the appropriate methods within the class is not possible.
  - How can access to an object of this class be synchronized?
    - Fortunately, the solution to this problem put calls to the methods defined by this class inside a synchronized block.

```
public void run(){  
    synchronized(objref){  
        ...  
    }  
}
```

```

class Counter implements Runnable{
    private int c = 0;
    public void increment() {
        try {
            c++;
            Thread.sleep(1000);
        } catch (InterruptedException e) { e.printStackTrace(); }

    }
    public void decrement() {
        try {
            c--;
            Thread.sleep(1000);
        } catch (InterruptedException e) { e.printStackTrace(); }

    }
    public int getValue() {
        return c; }
    public synchronized void call(){
        for (int i = 0; i < 2; i++) {
            this.increment();
            System.out.println("Value After increment "+ Thread.currentThread().getName() + " " + this.getValue());
            this.decrement();
            System.out.println("Value after decrementing:"+ Thread.currentThread().getName()+" " +this.getValue());
        }
    }
    public void run() {
        //synchronized(this) {
        this.call();
        //}
    }
}

```

```

public class Thread2 {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(counter, "Thread-1");
        Thread t2 = new Thread(counter, "Thread-2");
        Thread t3 = new Thread(counter, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}

```

```

Value After increment Thread-1 1
Value after decrementing Thread-1 0
Value After increment Thread-1 1
Value after decrementing Thread-1 0
Value After increment Thread-3 1
Value after decrementing Thread-3 0
Value After increment Thread-3 1
Value after decrementing Thread-3 0
Value After increment Thread-2 1
Value after decrementing Thread-2 0
Value After increment Thread-2 1
Value after decrementing Thread-2 0

```

# Interthread communication

- To avoid polling, Java includes an elegant interthread communication mechanism via the `wait( )`, `notify( )`, and `notifyAll( )` methods.

```
final void wait( ) throws InterruptedException  
final void notify( )  
final void notify All( )
```

- All three methods can be called only from within a synchronized context.
- `wait( )` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify( )` or `notifyAll( )`.
- `notify( )` wakes up a thread that called `wait( )` on the same object.
- `notifyAll( )` wakes up all the threads that called `wait( )` on the same object. One of the threads will be granted access.

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
```

```
class Producer implements Runnable {
    Q q;
    Thread t;

    Producer(Q q) {
        this.q = q;
        t = new Thread(this, "Producer");
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Output:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

# Interthread communication

- Inside `get( )`, `wait( )` is called. This causes its execution to suspend until Producer notifies you that some data is ready.
- When this happens, execution inside `get( )` resumes.
- After the data has been obtained, `get( )` calls `notify( )`.
- This tells Producer that it is okay to put more data in the queue.
- Inside `put( )`, `wait( )` suspends execution until Consumer has removed the item from the queue.
- When execution resumes, the next item of data is put in the queue, and `notify( )` is called.
- This tells Consumer that it should now remove it.



```

class Counter {
    private int c = 0;
    private boolean flag = false;
    public synchronized void increment() {
        while (flag) {
            try {
                System.out.println(Thread.currentThread().getName() + " I am waiting");
                wait();
            } catch (InterruptedException e) { e.printStackTrace(); }
        }
        for (int i = 0; i < 3; i++)
            c++;
        flag = true;
        System.out.println("incremented value:" + this.getValue());
        notify();
    }
    public synchronized void decrement() {
        while (!flag) {
            try {
                System.out.println(Thread.currentThread().getName() + " I am waiting");
                wait();
            } catch (InterruptedException e) { e.printStackTrace(); }
        }
        for (int i = 0; i < 3; i++)
            c--;
        System.out.println("decremented value:" + this.getValue());
        flag = false;
        notify();
    }
    public int getValue() {
        return c;
    }
}

```

```

class incremter implements Runnable{
    Counter count;
    incremter(Counter count){
        this.count=count;
    }
    public void run() {
        int i=0;
        while(i<5){
            count.increment();
            i++; }
    }
}

class decremter implements Runnable {
    Counter count;
    decremter(Counter count) {
        this.count = count;
    }
    public void run() {
        int i = 0;
        while (i < 5) {
            count.decrement();
            i++;
        }
    }
}

public class Thread2 {
    public static void main(String[] args) {
        Counter count=new Counter();
        incremter inc = new incremter(count);
        decremter dec = new decremter(count);
        Thread t1 = new Thread(inc, threadName: "Thread-Increment");
        Thread t2 = new Thread(dec, threadName: "Thread-Decrement");
        t1.start();
        t2.start();
    }
}

```

## Output

```

incremented value:3
Thread-Increment I am waiting
decremented value:0
Thread-Decrement I am waiting
incremented value:3
Thread-Increment I am waiting
decremented value:0
Thread-Decrement I am waiting
incremented value:3
Thread-Increment I am waiting
decremented value:0
Thread-Decrement I am waiting
incremented value:3
Thread-Increment I am waiting
decremented value:0

```

// An example of deadlock.

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();

        System.out.println(name + " entered A.foo");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("A Interrupted");
        }

        System.out.println(name + " trying to call B.last()");
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
```

```
            System.out.println("B Interrupted");
        }

        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside B.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Thread t;

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        t = new Thread(this, "RacingThread");
    }

    void deadlockStart() {
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }

    public void run() {
        b.bar(a); // get lock on b in other thread.
        System.out.println("Back in other thread");
    }

    public static void main(String args[]) {
        Deadlock dl = new Deadlock();

        dl.deadlockStart();
    }
}
```

MainThread entered A.foo  
RacingThread entered B.bar  
MainThread trying to call B.last()  
RacingThread trying to call A.last()

# Exchanger

- designed to simplify the exchange of data between two threads.
- Exchanger simply waits until two separate threads call its `exchange()` method. When that occurs, it exchanges the data supplied by the threads.
- For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection.
- The two threads work together so that each time a new buffer is needed, an exchange is made.

- Exchanger is a generic class that is declared as shown here:
- **Exchanger<V>**
- Here, V specifies the type of the data being exchanged.
- The only method defined by Exchanger is exchange( ), which has the two forms shown here:
- **V exchange(V objRef) throws InterruptedException**
- **V exchange(V objRef, long wait, TimeUnit tu) throws InterruptedException, TimeoutException**
- Here, objRef is a reference to the data to exchange.
- The second form of exchange( ) allows a time-out period to be specified.
- The key point about exchange( ) is that it won't succeed until it has been called on the same Exchanger object by two separate threads.
- Thus, exchange( ) synchronizes the exchange of the data.

```
// An example of Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
    public static void main(String args[]) {
        Exchanger<String> exgr = new Exchanger<String>();

        new Thread(new UseString(exgr)).start();
        new Thread(new MakeString(exgr)).start();
    }
}

// A Thread that constructs a string.
class MakeString implements Runnable {
    Exchanger<String> ex;
    String str;

    MakeString(Exchanger<String> c) {
        ex = c;
        str = new String();
    }

    public void run() {
        char ch = 'A';

        for(int i = 0; i < 3; i++) {

            // Fill Buffer
            for(int j = 0; j < 5; j++)
                str += ch++;

            try {
                // Exchange a full buffer for an empty one.
                str = ex.exchange(str);
            } catch (InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}
}
```

```
// A Thread that uses a string.
class UseString implements Runnable {
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
        ex = c;
    }

    public void run() {

        for(int i=0; i < 3; i++) {
            try {
                // Exchange an empty buffer for a full one.
                str = ex.exchange(new String());
                System.out.println("Got: " + str);
            } catch (InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}
}
```

## Output

```
Got: ABCDE
Got: FGHIJ
Got: KLMNO
```

# CyclicBarrier

- CyclicBarrier enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.
- CyclicBarrier has the following two constructors:
  - `CyclicBarrier(int numThreads)`
  - `CyclicBarrier(int numThreads, Runnable action)`
- Here, numThreads specifies the number of threads that must reach the barrier before execution continues.
- In the second form, action specifies a thread that will be executed when the barrier is reached.

# General procedure to use CyclicBarrier.

- First, create a CyclicBarrier object, specifying the number of threads that you will be waiting for.
- Next, when each thread reaches the barrier, have it call `await( )` on that object.
- will pause execution of the thread until all of the other threads also call `await( )`.
- Once the specified number of threads has reached the barrier, `await( )` will return and execution will resume.
- The `await( )` method has the following two forms:
  - `int await( )` throws `InterruptedException`, `BrokenBarrierException`
  - `int await(long wait, TimeUnit tu)` throws `InterruptedException`, `BrokenBarrierException`, `TimeoutException`
- The first form waits until all the threads have reached the barrier point. The second form waits only for the period of time specified by `wait`.



```
// An example of CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

        System.out.println("Starting");

        new Thread(new MyThread(cb, "A")).start();
        new Thread(new MyThread(cb, "B")).start();
        new Thread(new MyThread(cb, "C")).start();

    }
}
```

```
class BarAction implements Runnable {
    public void run() {
        System.out.println("Barrier Reached!");
    }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
```

```
// A thread of execution that uses a CyclicBarrier

class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;
        name = n;
    }

    public void run() {

        System.out.println(name);

        try {
            cbar.await();
        } catch (BrokenBarrierException exc) {
            System.out.println(exc);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}

// An object of this class is called when the
// CyclicBarrier ends.
```



```
public static void main(String args[]) {  
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );  
  
    System.out.println("Starting");  
  
    new Thread(new MyThread(cb, "A")).start();  
    new Thread(new MyThread(cb, "B")).start();  
    new Thread(new MyThread(cb, "C")).start();  
    new Thread(new MyThread(cb, "X")).start();  
    new Thread(new MyThread(cb, "Y")).start();  
    new Thread(new MyThread(cb, "Z")).start();  
  
}
```

A CyclicBarrier can be reused because it will release waiting threads each time the specified number of threads calls await( ). For example, if you change main( ) in the preceding program so that it looks like this:

the following output will be produced. (The precise order in which the threads execute may vary.)

```
Starting  
A  
B  
C  
Barrier Reached!  
X  
Y  
  
Z  
Barrier Reached!
```