



# MULTITHREADED PROGRAMMING IN JAVA



# Introduction(1)

Java provides built-in support for multithreaded programming.

- ❖ A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multithreading is a specialized form of multitasking.

There are two distinct types of multitasking:

1. process-based and
  2. thread-based.
- Process-based multitasking is the feature that allows computer to run two or more programs concurrently.
  - In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
  - ✓ In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
  - ✓ This means that a single program can perform two or more tasks simultaneously.
  - ✓ Multitasking threads require less overhead than multitasking processes.
  - Processes are heavyweight tasks that require their own separate address spaces.
  - ✓ Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.



# Introduction(2)

- Interprocess communication is expensive and limited.
- ✓ Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.
- ✓ Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- ✓ Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- ✓ Multithreading gain access to this idle time and put it to good use.



# Java Thread Model

- Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.
- Single-threaded systems use an approach called an event loop with polling.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.
- One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.
- Threads exist in several states.
  - A thread can be running. It can be ready to run as soon as it gets CPU time.
  - A running thread can be suspended, which temporarily suspends its activity.
  - A suspended thread can then be resumed, allowing it to pick up where it left off.
  - A thread can be blocked when waiting for a resource.
  - At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.



# Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

**The rules that determine when a context switch takes place are simple:**

1. A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
2. A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.





# Synchronization

- ❖ if two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it.
- ✓ For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor.
- ❖ The monitor is a control mechanism . Monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor.
- ✓ **Java provides a cleaner solution.** There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called.
- ✓ Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.



# Messaging

- After you divide your program into separate threads, you need to define how they will communicate with each other.
- When programming with most other languages, you must depend on the operating system to establish communication between threads.
- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects 'have.
- Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.



# Thread Class

- ❖ Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.
- To create a new thread, your program will either extend Thread or implement the Runnable interface.
- The Thread class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.





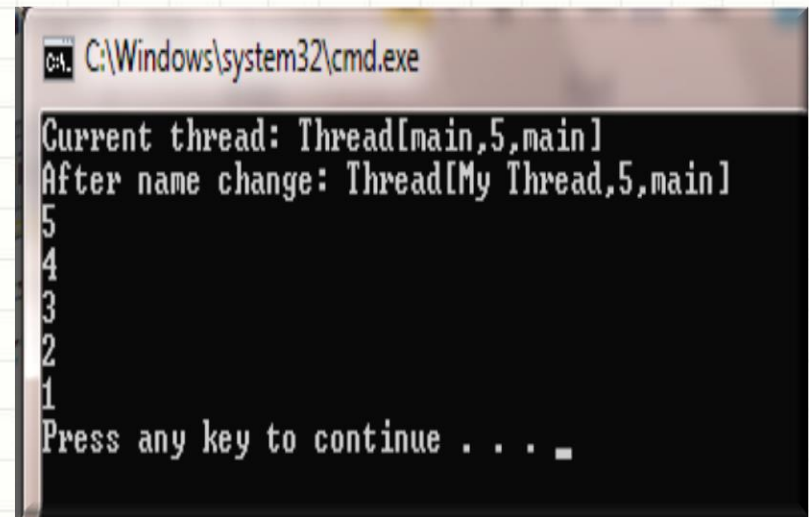
# Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.

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.

```
class CurrentThreadDemo {  
public static void main(String args[]) {  
    Thread t = Thread.currentThread();  
    System.out.println("Current thread: " + t);  
    t.setName("My Thread");  
  
    System.out.println("After name change: " + t);  
    try {  
        for(int n = 5; n > 0; n--) {  
            System.out.println(n);  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println("Main thread interrupted");  
    }  
}
```



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output of the Java program is displayed as follows:

```
Current thread: Thread[main,5,main]  
After name change: Thread[My Thread,5,main]  
5  
4  
3  
2  
1  
Press any key to continue . . .
```



# Creating a Thread

Java defines two ways in which this can be achieved:

1. By implementing the Runnable interface.
2. By extending the Thread class, itself.

**class NewThread implements Runnable {**  
**Thread t;**

**NewThread() {**  
**t = new Thread(this, "Demo Thread");**  
**System.out.println("Child thread: " + t);**  
**t.start(); // Start the thread**  
**}**

**public void run() {**  
**try {**  
**for (int i = 5; i > 0; i--) {**  
**System.out.println("Child Thread: " + i);**  
**Thread.sleep(500);**  
**}**  
**} catch (InterruptedException e) {**  
**System.out.println("Child interrupted.");**  
**}**  
**System.out.println("Exiting child thread.");**  
**}**

```
C:\Windows\system32\cmd.exe
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
Press any key to continue . . .
```

**public class ThreadMain {**  
**public static void main(String args[]) {**  
**new NewThread(); // create a new thread**  
**try {**  
**for (int i = 5; i > 0; i--) {**  
**System.out.println("Main Thread: " + i);**  
**Thread.sleep(1000);**  
**}**  
**} catch (InterruptedException e) {**  
**System.out.println("Main thread**  
**interrupted.");**  
**}**  
**System.out.println("Main thread exiting.");**  
**}**  
**}**



# How to choose

**why Java has two ways to create child threads and which approach is better ?**

- The Thread class defines several methods that can be overridden by a derived class.
- Of these methods, the only one that must be overridden is run( ).
- This is, of course, the same method required when you implement Runnable.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way.
- So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- Finally choice up to you.



```

class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

```

```

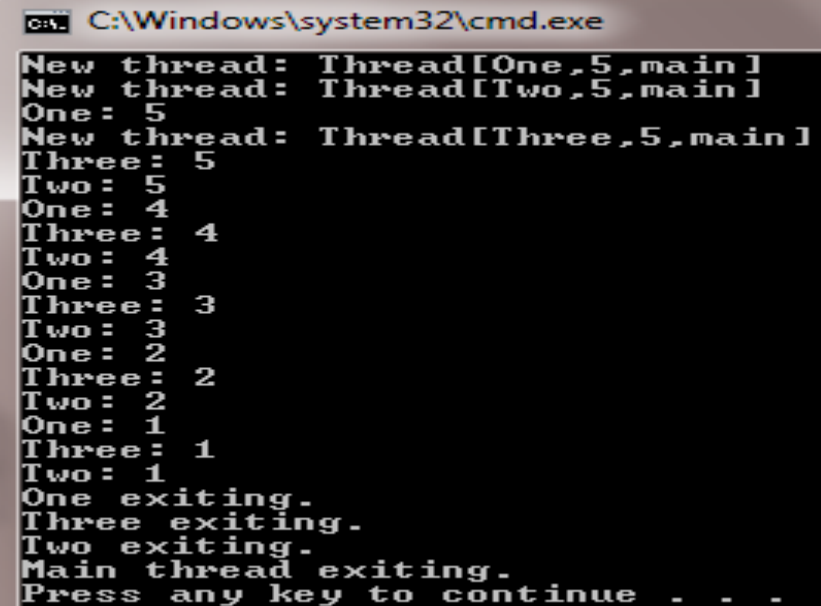
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name +
                "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

```

```

public class MultiThreadMain {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread
                Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}

```



```

C:\Windows\system32\cmd.exe
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Three: 5
Two: 5
One: 4
Three: 4
Two: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Three exiting.
Two exiting.
Main thread exiting.
Press any key to continue . . .

```



# Using `isAlive()` and `join()`

How can one thread know when another thread has ended?

Two ways exist to determine whether a thread has finished.

First, by calling ***isAlive()*** on the thread. This method is defined by Thread, and its general form is shown here:

```
final boolean isAlive( )
```

The `isAlive()` method returns true if the thread upon which it is called is still running. It returns false otherwise.

While `isAlive()` is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called ***join()***, shown here:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates.





```
class NewThread implements Runnable {  
    String name;  
    Thread t;  
    NewThread(String threadname) {  
        name = threadname;  
        t = new Thread(this, name);  
        System.out.println("New thread: " + t);  
        t.start();  
    }  
    public void run() {  
        try {  
            for (int i = 5; i > 0; i--) {  
                System.out.println(name + ": " + i);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println(name + " interrupted.");  
        }  
        System.out.println(name + " exiting.");  
    }  
}  
  
public class ThreadJoinMain {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
        NewThread ob3 = new NewThread("Three");  
        System.out.println("Thread One is alive: " +  
            ob1.t.isAlive());
```

```
        System.out.println("Thread Two is alive: " +  
            ob2.t.isAlive());  
        System.out.println("Thread Three is alive: " +  
            ob3.t.isAlive());  
        try {  
            System.out.println("Waiting for threads to  
                finish.");  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Main thread  
                Interrupted");  
        }  
        System.out.println("Thread One is alive: " +  
            ob1.t.isAlive());  
        System.out.println("Thread Two is alive: " +  
            ob2.t.isAlive());  
        System.out.println("Thread Three is alive: "  
            + ob3.t.isAlive());  
        System.out.println("Main thread exiting.");  
    }  
}
```



C:\WINDOWS\system32\cmd.exe

```
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
One exiting.
Three exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
Press any key to continue . . . _
```



# Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread can preempt a lower-priority one.
- To set a thread's priority, use the `setPriority( )` method, which is a member of `Thread`.

*This is its general form:*

**`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 return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5.



```
public class priority implements Runnable {
```

```
    @Override
```

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

```
        for (int x = 1; x <= 3; x++)
```

```
            System.out.println(x + " This is thread " + Thread.currentThread().getName());
```

```
    }
```

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

```
        Thread ta1 = new Thread(new priority(), "Thread A");
```

```
        Thread t2 = new Thread(new priority(), "Thread B");
```

```
        Thread t3 = new Thread(new priority(), "Thread C");
```

```
        t1.setPriority(10);
```

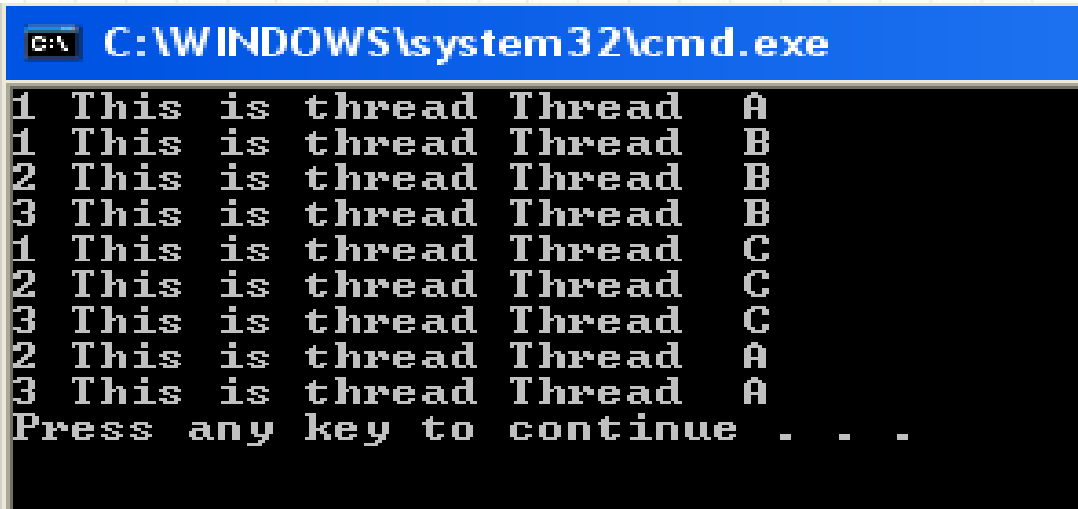
```
        t1.start();
```

```
        t2.start();
```

```
        t3.start();
```

```
    }
```

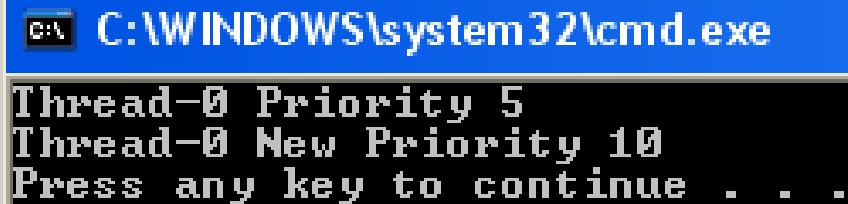
```
}
```



```
C:\WINDOWS\system32\cmd.exe
1 This is thread Thread A
1 This is thread Thread B
2 This is thread Thread B
3 This is thread Thread B
1 This is thread Thread C
2 This is thread Thread C
3 This is thread Thread C
2 This is thread Thread A
3 This is thread Thread A
Press any key to continue . . .
```



```
public class SetPriority implements Runnable {  
    Thread th;  
  
    public SetPriority() {  
        th = new Thread(this);  
        th.start();  
    }  
  
    public void run() {  
        //get Priority of the thread  
        System.out.println( th.getName()+" Priority "+th.getPriority());  
        //set Priority of the thread  
        th.setPriority(Thread.MAX_PRIORITY);  
        //get Priority of the thread  
        System.out.println(th.getName()+" New Priority "+th.getPriority());  
    }  
  
    public static void main(String[] args) {  
        new SetPriority();  
    }  
}
```



C:\WINDOWS\system32\cmd.exe

Thread-0 Priority 5  
Thread-0 New Priority 10  
Press any key to continue . . .





# 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.
- ❖ Java provides unique, language-level support for it.
  - Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.

You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword.

## Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.



```

class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}

```

```

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target,
        "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

C:\WINDOWS\system32\cmd.exe

[Hello[World[Synchronized]

]

]

Press any key to continue . . .



# Inter Thread Communication

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

These methods are implemented as final methods in `Object`, so all classes have them. 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()`.
- `notify()` wakes up the first thread that called `wait()` on the same object.
- `notifyAll()` wakes up all the threads that called `wait()` on the same object.
- The highest priority thread will run first.



```

public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}

class CubbyHole {
    private int contents;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        available = false;
        notifyAll();
        return contents;
    }
    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        contents = value;
        available = true;
        notifyAll();
    }
}

```

```

class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}

class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" +
                               this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {}
        }
    }
}

```



C:\WINDOWS\system32\cmd.exe

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
Press any key to continue . . .
```





```

class NewThread implements Runnable {
String name; // name of thread
Thread t;
boolean suspendFlag;

NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
suspendFlag = false;
t.start(); // Start the thread    }

public void run() {
try {
for(int i = 15; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);
synchronized(this) {
while(suspendFlag) {
wait();    }    }    }
catch (InterruptedException e) {
System.out.println(name + " interrupted.");
System.out.println(name + " exiting.");    }
void mysuspend() {
suspendFlag = true;    }
synchronized void myresume() {
suspendFlag = false;
notify();    }
}
}

```

```

class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {
Thread.sleep(1000);
ob1.mysuspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
System.out.println("Main thread exiting.");
}
}
}

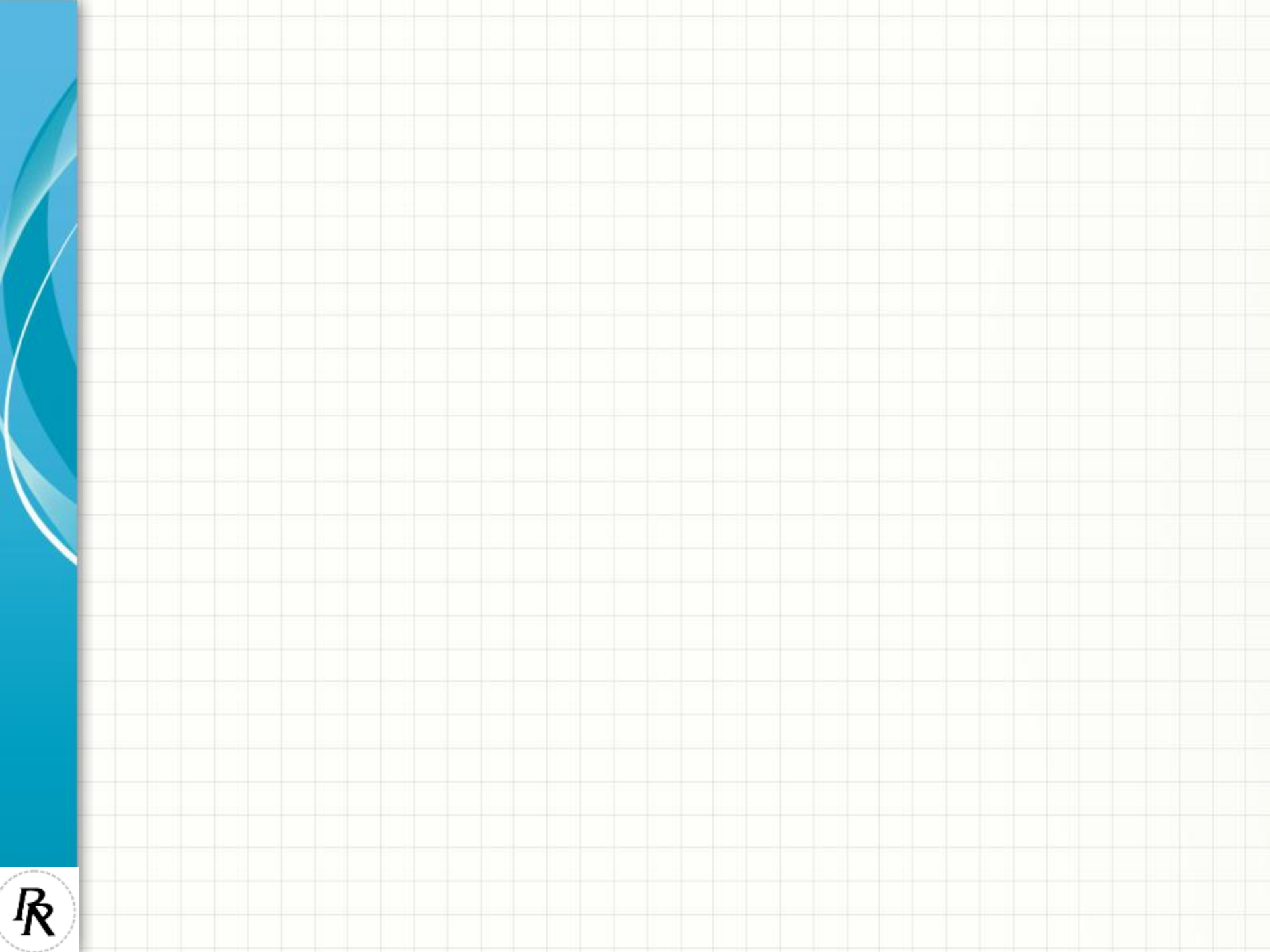
```



C:\WINDOWS\system32\cmd.exe

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 15
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
Press any key to continue . . . _
```





# Enumerations, Autoboxing



# Enumerations

- Enumeration is a list of named constants.
- In languages such as C++, enumerations are simply lists of named integer constants.
- In Java, an enumeration defines a class type.
- Enumeration can have constructors, methods, and instance variables.





# Enumeration Fundamentals

- An enumeration is created using the enum keyword.

// An enumeration of apple varieties.

```
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap,  
    Cortland  
}
```



- The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
- Each is implicitly declared as a public, static final member of Apple.
- Their type is the type of the enumeration in which they are declared, which is Apple.
- Once we have defined an enumeration, we can create a variable of that type.



- Even though its of class type, no need to instantiate an enum using new.

- We can declare and use an enumeration variable.

Eg: Apple ap;

- ap is of type Apple, so it can be assigned the value only defined by the enumeration.

Eg: ap = Apple.RedDel;

- ap is having the value RedDel:



- To display an enumeration constant using `println( )` statement.

Eg: `System.out.println(Apple.Winesap);`

Output :

Winesap



# The values( ) and valueOf( ) Methods

- All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**.

General forms :

```
public static enum-type[ ] values( )
```

```
public static enum-type valueOf(String str)
```

- **values( )** method returns an array that contains a list of the enumeration constants.
- **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in str.





```
// Use the built-in enumeration methods.
// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[ ])
    {
        Apple ap;
        Apple allapples[ ] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);
        System.out.println();
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```



Output :

Jonathan

GoldenDel

RedDel

Winesap

Cortland

ap contains Winesap



# Java Enumerations Are Class Types

- Java enumeration is a class type.
- Even though we don't instantiate an enum using new, it has the same capabilities as other classes.
- It can have constructors, instance variables and methods.
- Implement interfaces.



```
enum Apple {  
    Jonathan(10), GoldenDel(9), RedDel(12),  
    Winesap(15), Cortland(8);  
    private int price; // price of each apple  
    // Constructor  
    Apple(int p) { price = p; }  
    int getPrice() { return price; }  
}  
  
class EnumDemo3 {  
    public static void main(String args[])  
    {  
        Apple ap;  
        // Display price of Winesap.  
        System.out.println("Winesap costs " +  
            Apple.Winesap.getPrice() + " cents.\n");  
    }  
}
```



Output:

Winesap costs 15 cents.





# Enumerations Inherit Enum

- All enumerations automatically inherit `java.lang.Enum`.
- This class defines several methods that are available for use by all enumerations.
- `ordinal( )`, `compareTo( )`, `equals( )`.



- **final int ordinal( )**
  - It returns the ordinal value of the invoking constant. i.e; position of enumeration constant in the list.
- **final int compareTo(enum-type e)**
  - Compares the ordinal value of two constants.
- **equals( )**
  - Compares two enumeration constant. If they are equal it returns true.



```
// Demonstrate ordinal(), compareTo() and equals().
// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland }
class EnumDemo4 {
    public static void main(String args[])
    {
        Apple ap, ap2, ap3;
        // Obtain all ordinal values using ordinal().
        System.out.println("Apple constants" +
                           " and their ordinal values: ");
        for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());
        ap = Apple.RedDel;
        ap2 = Apple.GoldenDel;
        ap3 = Apple.RedDel;
        System.out.println();
    }
}
```



```
// Demonstrate compareTo() and equals()  
if(ap.compareTo(ap2) < 0)  
    System.out.println(ap + " comes before "  
                        + ap2);  
  
if(ap.equals(ap3))  
    System.out.println(ap + " equals " + ap3);  
if(ap.equals(ap2))  
    System.out.println("Error!");  
}  
}
```



# Output:

Apple constants and their ordinal values:

Jonathan 0

GoldenDel 1

RedDel 2

Winesap 3

Cortland 4

GoldenDel comes before RedDel

RedDel equals RedDel





# Autoboxing

- Two important features: autoboxing and auto-unboxing.
- **Autoboxing** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- **Auto-unboxing** is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.



- In autoboxing its not necessary to manually construct an object to wrap a primitive type.
- We need to only assign that value to a type-wrapper reference.
- Eg: To construct an Integer object that has the value 100:

```
Integer iOb = 100; // autobox an int
```



- To unbox an object, simply assign that object reference to a primitive-type variable.

Eg: To unbox iOb

```
int i = iOb; // auto-unbox
```



// Demonstrate autoboxing/unboxing.

```
class AutoBox {  
    public static void main(String args[ ]) {  
        Integer iOb = 100; // autobox an int  
        int i = iOb; // auto-unbox  
        System.out.println(i + " " + iOb);  
    }  
}
```

**Output:**

100 100



# Autoboxing and Methods

- Autoboxing automatically occurs whenever a primitive type must be converted into an object.
- Auto-unboxing takes place whenever an object must be converted into a primitive type.
- Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.



```
class AutoBox2 {  
    // Take an Integer parameter and return an int  
    value  
    static int m(Integer v) {  
        return v ; // auto-unbox to int  
    }  
  
    public static void main(String args[]) {  
  
        Integer iOb = m(100);  
        System.out.println(iOb);  
    }  
}
```





# Autoboxing/Unboxing Occurs in Expressions

Eg:

```
Integer iOb, iOb2;
```

```
int i;
```

```
iOb = 100;
```

```
iOb2 = iOb + (iOb / 3); //autobox
```

```
i = iOb + (iOb / 3); //unbox
```



- Auto-unboxing also allows to mix different types of numeric objects:

Eg:

Integer iOb = 100;

Double dOb = 98.6;

dOb = dOb + iOb;

# Autoboxing/Unboxing Boolean and Character Values

Eg:

```
Boolean b = true;//autobox  
if(b) //unbox  
System.out.println("b is true");
```

Eg: Autobox/unbox a char.

```
Character ch = 'x'; // box a char  
char ch2 = ch; // unbox a char
```





## Unit 4....