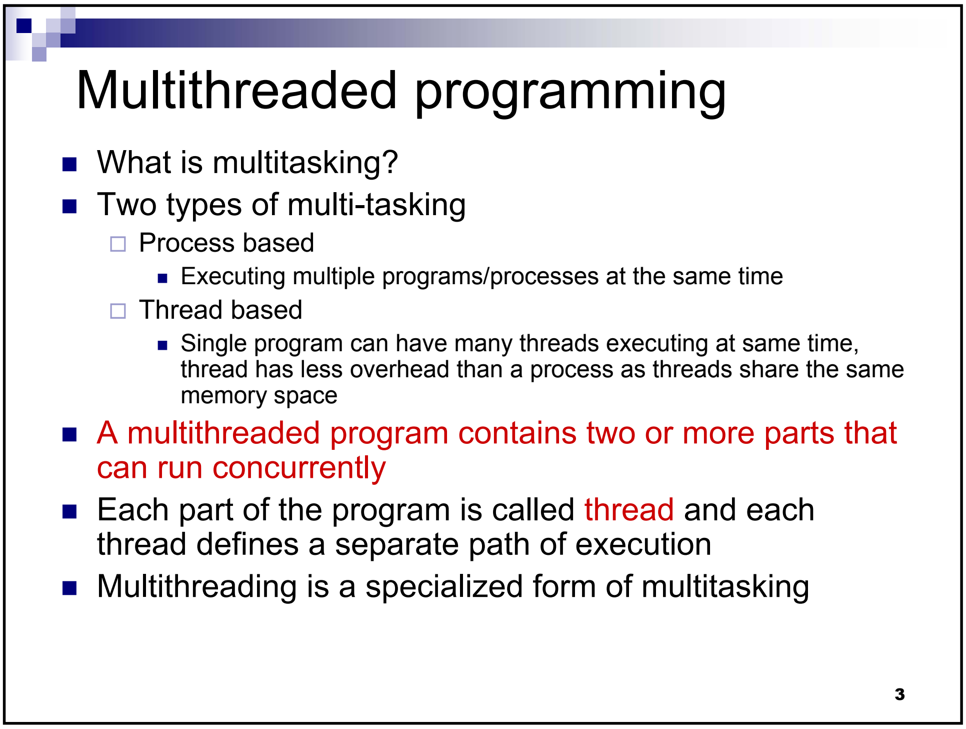
Threads

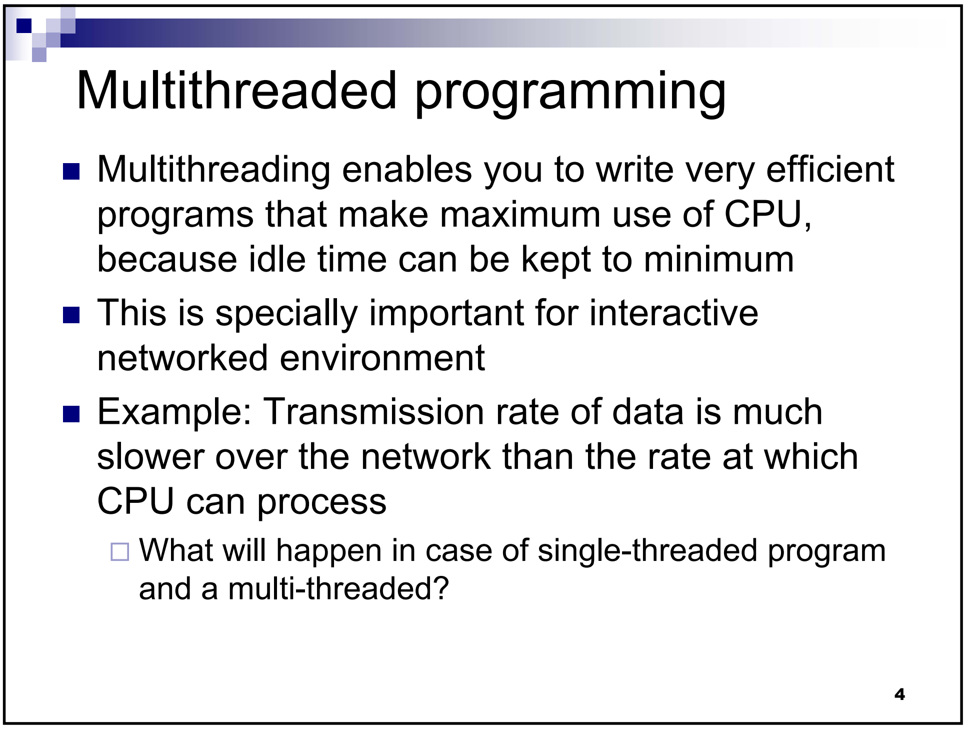
Topics

- Multithreaded programming
- Thread Lifecycle
- Thread Class
- Runnable Interface
- Main Thread
- Thread Priorities
- Synchronization
- Messaging



Multithreaded programming

- What is multitasking?
- Two types of multi-tasking
  - Process based
    - Executing multiple programs/processes at the same time
  - Thread based
    - Single program can have many threads executing at same time, thread has less overhead than a process as threads share the same memory space
- A multithreaded program contains two or more parts that can run concurrently
- Each part of the program is called thread and each thread defines a separate path of execution
- Multithreading is a specialized form of multitasking



Multithreaded programming

- Multithreading enables you to write very efficient programs that make maximum use of CPU, because idle time can be kept to minimum
- This is specially important for interactive networked environment
- Example: Transmission rate of data is much slower over the network than the rate at which CPU can process
  - What will happen in case of single-threaded program and a multi-threaded?

Thread Lifecycle

- Thread exists in several states
- A thread can be *running*
- It can be *ready* to run as soon as it gets the CPU
- A running thread can be *suspended*, which temporarily suspends its activity
- 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 it cannot be resumed

Thread class and **Runnable** Interface

- Java’s multithreaded system is built upon the **Thread** class, it methods and its companion interface **Runnable**.
- To create a thread your program will either extend **Thread** or implement **Runnable**.

The Main Thread

- When a java program starts, 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 a thread from which other child threads can be spawned
  - □ Often it is the last thread to finish execution because it performs various shutdown activities

The Main Thread

- Although the main thread starts automatically when the program is started, it can be controlled through a **Thread** object.
- We can do this by calling the currentThread() method, which is a public static member of class **Thread**
- This method returns a reference to the thread in which it is called. Once you have the reference you can control the thread like any other thread.

Example

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

            Thread t = Thread.currentThread();
            System.out.println("Current Thread: " + t);

            //change the name of current thread
            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 ie){
                        System.out.println("Main thread interrupted!");
            }
            }
}
```

Example output

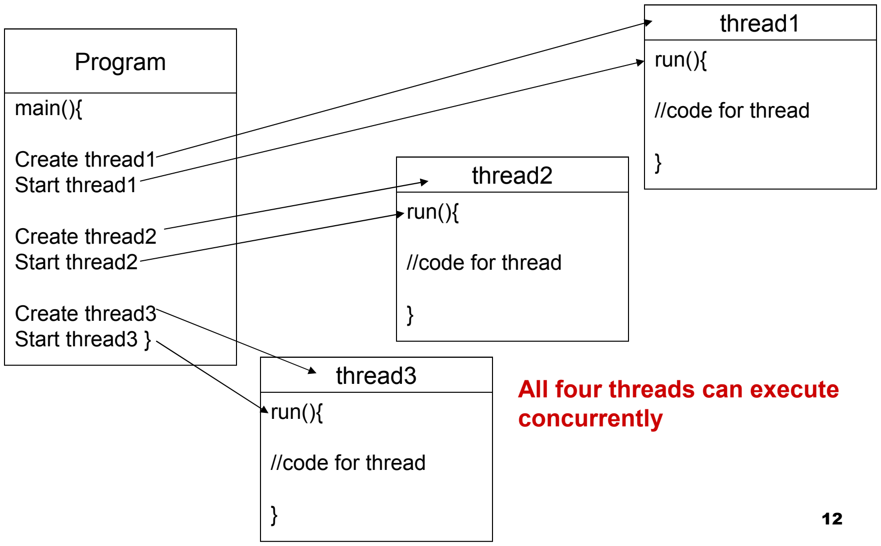
```
Current Thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

- **The System.out.println() method displays the Thread Name, its priority and the group**
- **A *Thread Group* is a data structure that controls the state of a collection of thread as a whole**

Creating a Thread

- We can implement **Runnable**
- or extend the **Thread** class

Creating Thread



Program

main(){

Create thread1
Start thread1

Create thread2
Start thread2

Create thread3
Start thread3 }

thread1

run(){

//code for thread

}

thread2

run(){

//code for thread

}

thread3

run(){

//code for thread

}

All four threads can execute concurrently

Extending Thread Example

```
public class TryThread extends Thread {
  public TryThread(String firstName, String secondName, long delay) {
    this.firstName = firstName;      // Store the first name
    this.secondName = secondName;    // Store the second name
    aWhile = delay;                  // Store the delay
    setDaemon(true);                 // Thread is daemon
  }

  public static void main(String[] args) {
    // Create three threads
    Thread first = new TryThread("Hopalong ", "Cassidy ", 200L);
    Thread second = new TryThread("Marilyn ", "Monroe ", 300L);
    Thread third = new TryThread("Slim ", "Pickens ", 500L);

    System.out.println("Press Enter when you have had enough...\n");
    first.start();                   // Start the first thread
    second.start();                  // Start the second thread
    third.start();                   // Start the third thread
```

Extending Thread Example

```
try {
      System.in.read();              // Wait until Enter key pressed
      System.out.println("Enter pressed...\n");

    } catch (IOException e) {        // Handle IO exception
      System.out.println(e);         // Output the exception
    }
    System.out.println("Ending main()");
    return;
  }
                                             // Method where thread execution will start
                                               public void run() {
                                                 try {
                                                   while(true) {                    // Loop indefinitely...
                                                     System.out.print(firstName);       // Output first name
                                                     sleep(aWhile);                 // Wait aWhile msec.
                                                     System.out.print(secondName + "\n"); // Output second name
                                                   }
                                                 } catch(InterruptedException e) {      // Handle thread interruption
                                                   System.out.println(firstName + secondName + e);    // Output the exception
                                                 }
                                               }

                                               private String firstName;        // Store for first name
                                               private String secondName;        // Store for second name
                                               private long aWhile;             // Delay in milliseconds
                                             }
```

Daemon and user threads

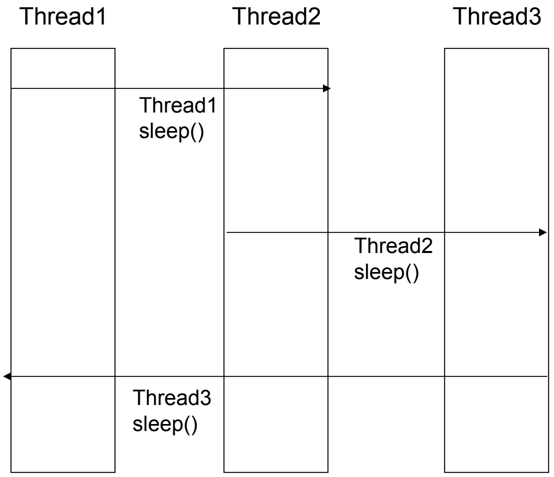
- A Daemon thread is simply a background thread that is subordinate to the thread that creates it
- When the thread that created daemon ends the daemon thread also ends and dies with it
- A thread is made daemon by the setDaemon() method
- A thread that is not daemon is called a user thread
- A user thread has a life of its own and it is not dependent of the thread that creates it

Implementing Runnable

```
import java.io.IOException;

public class JumbleNames implements Runnable {
  // Constructor
  public JumbleNames(String firstName, String secondName, long delay) {
    this.firstName = firstName;                    // Store the first name
    this.secondName = secondName;                  // Store the second name
    aWhile = delay;                                // Store the delay
  }

  // Method where thread execution will start
  public void run() {
    try {
      while(true) {                                // Loop indefinitely...
        System.out.print(firstName);               // Output first name
        Thread.sleep(aWhile);                      // Wait aWhile msec.
        System.out.print(secondName+"\n");         // Output second name
      }
    } catch(InterruptedException e) {              // Handle thread interruption
      System.out.println(firstName + secondName + e);    // Output the exception
    }
  }
public static void main(String[] args) {
  // Create three threads
  Thread first = new Thread(new JumbleNames("Hopalong ", "Cassidy ", 200L));
  Thread second = new Thread(new JumbleNames("Marilyn ", "Monroe ", 300L));
  Thread third = new Thread(new JumbleNames("Slim ", "Pickens ", 500L));

  // Set threads as daemon
  first.setDaemon(true);
  second.setDaemon(true);
  third.setDaemon(true);
  System.out.println("Press Enter when you have had enough...\n");
  first.start();                          // Start the first thread
  second.start();                         // Start the second thread
  third.start();                          // Start the third thread
  try {
    System.in.read();                     // Wait until Enter key pressed
    System.out.println("Enter pressed...\n");

  } catch (IOException e) {               // Handle IO exception
    System.out.println(e);                // Output the exception
  }
  System.out.println("Ending main()");
  return;
 }

 private String firstName;                // Store for first name
 private String secondName;               // Store for second name
 private long aWhile;                     // Delay in milliseconds
}
```

Thread Scheduling

- Preemptive Multitasking?



Thread1

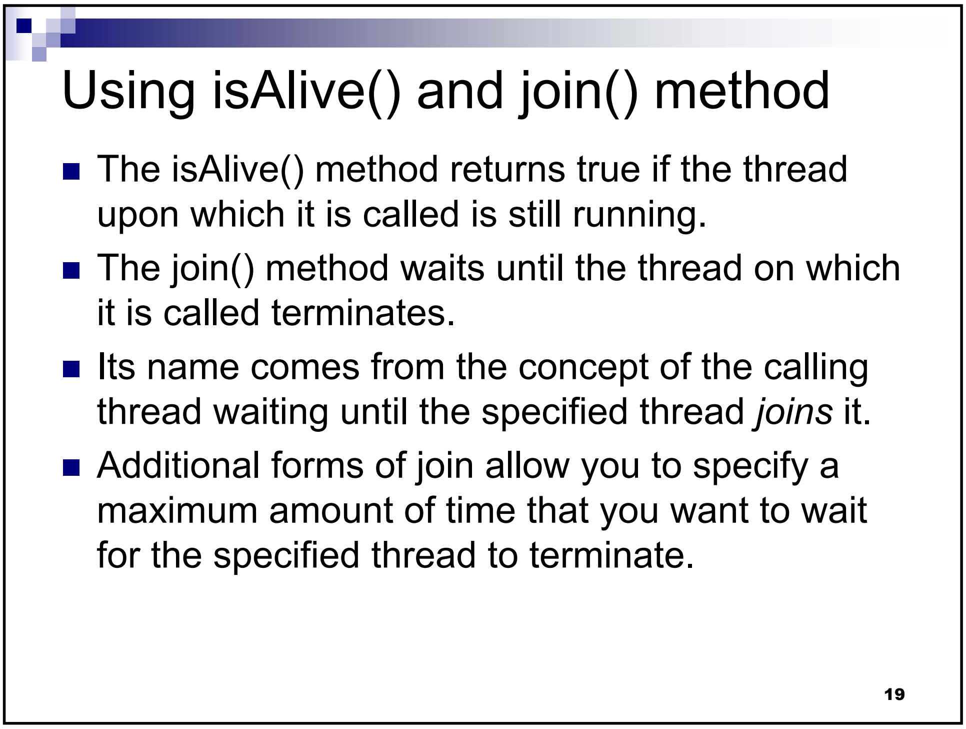
Thread2

Thread3

Thread1 sleep()

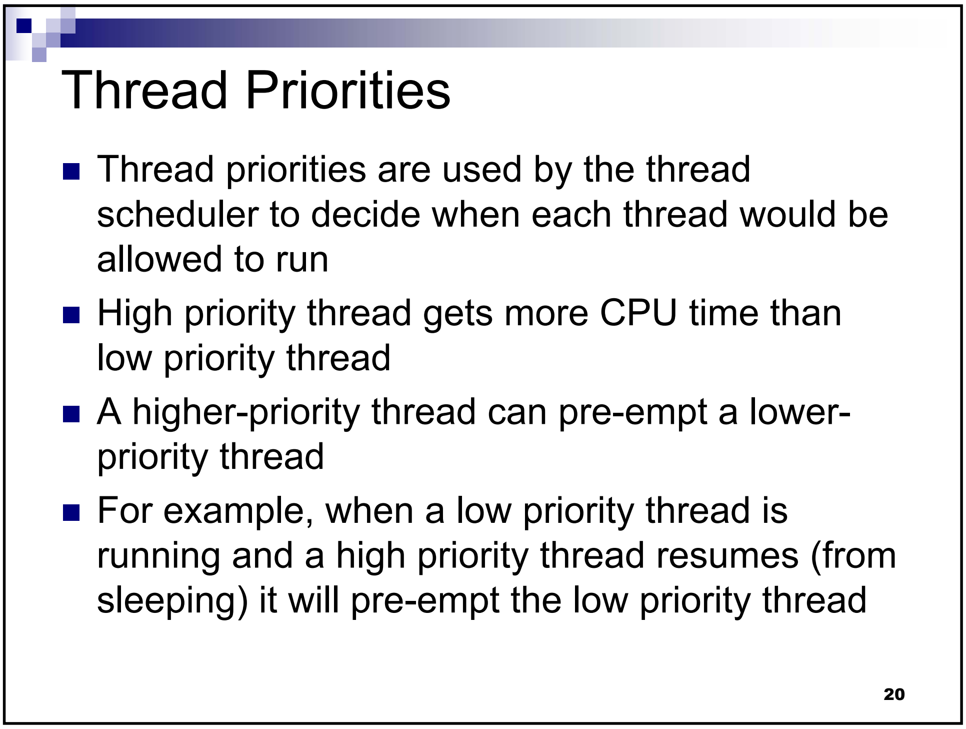
Thread2 sleep()

Thread3 sleep()



Using isAlive() and join() method

- The isAlive() method returns true if the thread upon which it is called is still running.
- The join() method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of join allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.



Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread would be allowed to run
- High priority thread gets more CPU time than low priority thread
- A higher-priority thread can pre-empt a lower-priority thread
- For example, when a low priority thread is running and a high priority thread resumes (from sleeping) it will pre-empt the low priority thread

Thread Priorities

- To set thread priority used the **setPriority(int level)** method which is a member of thread
- Level specifies the new priority setting for the calling thread
- The value is within the range MIN_PRIORITY to MAX_PRIORITY, these values are 1 and 10 respectively
- A level of 5 is default or NORM_PRIORITY
- These priorities are defines as **final** variables in **Thread**

Priority Example

```
class Clicker implements Runnable{

        long click=0;
        Thread t;
        private volatile boolean running=true;

        public Clicker(int p){
                t=new Thread(this);
                t.setPriority(p);
        }

        public void run(){
                while (running)
                        click++;
        }

        public void stop(){
                running=false;
        }

        public void start(){
                t.start();
        }
}
```

Priority Example

```
public class HiLo{

public static void main(String args[]){

          Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
          Clicker hi = new Clicker(Thread.NORM_PRIORITY+2);
          Clicker lo = new Clicker(Thread.NORM_PRIORITY-2);

          lo.start();
          hi.start();

          try{

          Thread.sleep(10000);
          }catch (InterruptedException ie){
                    System.out.println("Main thread interrupted!");
          }

          lo.stop();
          hi.stop();

          try{

                    hi.t.join();
                    lo.t.join();
          }catch (InterruptedException ie){
                    System.out.println("Exception Caught!");
          }

System.out.println("Low priority thread " + lo.click);
System.out.println("High priority thread " + hi.click);

}
}
```

Output

```
Low priority thread 36654484
High priority thread 2034854108
Press any key to continue...
```

Synchronization

- When two or more threads try to access the same 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 a monitor (or semaphore).
- A monitor is an object that is used as a mutually exclusive *lock*.

- Only one thread can own a monitor at one time.
- When a thread acquired a monitor it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor are *suspended* until the first thread *exists* the monitor.
- These other threads are said to be *waiting* for the monitor.

Using Synchronized Methods

- You make methods mutually exclusive by declaring them in the class using the keyword **synchronized**
- While a thread is inside the synchronized method, all the other threads trying to call it on the same instance have to wait
- Only when the currently executing synchronized method for an object has ended can another synchronized method start for the same object

Example

```
class HeatSync {
       private int[] intArray = new int[10];
       synchronized void reverseOrder() {
       int halfWay = intArray.length / 2;
       for (int i = 0; i < halfWay; ++i) {
        int upperIndex = intArray.length - 1 - i;
        int save = intArray[upperIndex];
       intArray[upperIndex] = intArray[i];
        intArray[i] = save;
     }
 }
```

Using **synchronized** blocks

- In addition to being able to synchronize methods on a class object, you can also specify a statement or a block of code in your program as **synchronized**
- This is more powerful since you specify which particular object is to benefit from synchronization of the statement or code block, not just the object that contains the synchronized method
- Here we can set a lock on any object for a given statement block
- When the block that is synchronized is executing, no other block or method that is synchronized on the same object can execute.
- No other statements or statement blocks in the program that are synchronized on the object can execute while the statement is executing

Using **synchronized** blocks

```
public void static absolute(int [] values) {
    synchronized (values) {
    for(int i=0; i < values.length; i++) {
    if(values[i] < 0)
    values[i] = -values[i];
    }
    }
}
```