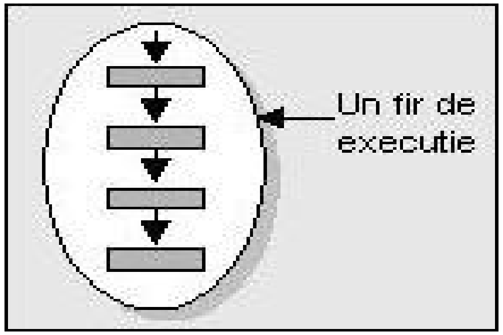
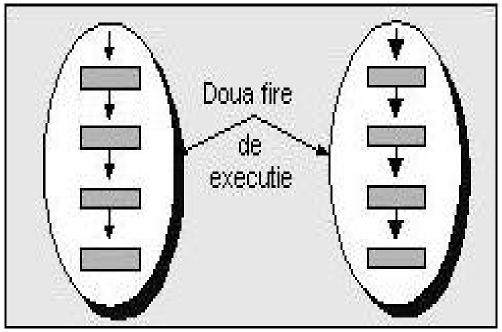
Advanced Programming
Concurrency

Concurrent Programming

- Until now, a program was a **sequence** of operations, executing one after another.
- In a **concurrent** program, *several* sequences of operations may execute **"in the same time"**, **interleaving** one with another.





- Advantages: background calculations, non-blocking IO, exploiting multi-core processors, etc.

  → High responsiveness

Threads

- The JVM runs as a single **process**. A process has a self-contained execution environment.
- The JVM allows an application to have multiple **threads** of execution running concurrently.
- **Threads exist within a process** - every process has at least one. Threads share the process's resources, including memory and open files.
- Creating a new thread (*lightweight process*) requires **fewer resources** than creating a new process.
- When a JVM starts up, there is usually a single thread, called **main**. When all threads have died, the JVM process stops.

Thread and *Runnable*

- Each executing thread is an instance of *java.lang.Thread* class or a subclass of it.
- A thread must “know” what code it is supposed to execute, so it will receive a *runnable object.*

```
Thread t = new Thread(Runnable target);
```

- The *java.lang.Runnable* interface should be implemented by any class whose instances are intended to be executed by a thread.

```
public void run() {
    // This is where we write the code executed by a thread
}
```

- *Thread* class already implements *Runnable*.

Creating and Running a Thread

```
// This thread writes important data to a file
public class HelloThread extends Thread {

  @Override
  public void run() {
    int count = 100_000_000;
    try (BufferedWriter out = new BufferedWriter(
          new FileWriter("hello.txt"))) {
      for (int i = 0; i < count; i++) {
        out.write("Hello World!\n");
      }
    } catch (IOException e) {
      System.err.println("Oops..." + e);
    }

    public static void main(String args[]) {
      // Start the thread
      new HelloThread().start();
      System.out.println("OK...");
    }
}
```

The *Thread.start* Method

```
new HelloThread().start();
```

- Causes the thread to begin execution.

  The JVM calls the **run** method of this thread.

- The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).
- It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

  → *IllegalThreadStateException*

Implementing *Runnable*

```
public class HelloRunnable implements Runnable {
  private final String filename;
  private final String message;
  private final int count;

  public HelloRunnable(String filename, String message, int count) {
    this.filename = filename;
    this.message = message;
    this.count = count;
  }

  @Override
  public void run() {
    // The same code as in the previous example . . .
  }

  public static void main(String args[]) {
    Runnable runnable = new HelloRunnable("hello.txt", "Ciao!", 10);
    new Thread(runnable).start();
  }
}
```

Implementing *Runnable* is more general and flexible than extending *Thread*, because the *Runnable* object can subclass a class other than *Thread*.

Using λ-expressions

Runnable is a **functional interface** (it has only one method)

```
public class TestRunnable {

    public static void main(String args[]) {
        TestRunnable app = new TestRunnable();
        app.testThreads();
    }

    private void testThreads() {
        //Define the runnable object using a lambda-expression
        Runnable runnable = () -> {
            System.out.println("Hello from thread 1");
        };
        new Thread(runnable).start();

        //Define the runnable object using a method reference
        new Thread(this::doSomething).start();
    }

    private void doSomething() {
        System.out.println("Hello from thread 2");
    }
}
```

Resource Contention

- Threads may run into conflicts over access to a **shared resource** such as memory, files, etc.

```
Runnable r1 = new HelloRunnable("hello.txt", "Hello World", 1000);
Runnable r2 = new HelloRunnable("hello.txt", "Ciao Mondo", 1000);
new Thread(r1).start();
new Thread(r2).start();
```

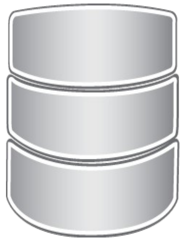
- What could happen when running the two threads?

```
Hello World
Hello World
Hello World
Hello World
…
Ciao Mondo
Ciao Mondo
Ciao Mondo
Ciao Mondo
Ciao Mondo
…
```

```
Ciao Mondo
Ciao Mondo
…
Ciao MonHello World
Hello World
Hello World
…
Hello Wodo
Ciao Mondo
Ciao Mondo
…
```

Thread Interference

- Threads communicate by reading/writing data from/to a **shared memory.**
  - Thread t1 ↔
  - Thread t2 ↔
  - ...



← Heap

- Operations on shared data might be interrupted mid-stream (**non-atomic**)
- **Interleaving:** two operations consist of multiple steps, and the sequences of steps overlap.
- **Memory consistency errors** occur when different threads have inconsistent views of what should be the same data

The Producer–Consumer Example

- Two threads, the *producer* and the *consumer*, share a common *buffer* and must *synchronize* their operations.
- The Buffer holds a number (or a string, array, etc.)

```
public class Buffer {
    private long number = -1;
    public long getNumber() {
        return number;
    }
    public void setNumber(long number) {
        this.number = number;
    }
}
```

- The main program starts the two threads:

```
    Buffer buffer = new Buffer();
    new Producer(buffer).start();
    new Consumer(buffer).start();
```

The Producer

```
//The producer generates numbers and puts them into the buffer
public class Producer extends Thread {

    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            buffer.setNumber(i);
            System.out.println("Number produced:" + i);

            try {
                sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
                System.err.println(e);
            }
        }
    }
}
```

The Consumer

```
//The consumer reads the numbers from the buffer
public class Consumer extends Thread {

    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
      for (int i = 0; i < 10; i++) {
        long value = buffer.getNumber();

        System.out.println(
         "Number consumed: " + value);
        }
    }
}
```

```
Number produced:0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number consumed: 0
Number produced:1
Number produced:2
Number produced:3
Number produced:4
Number produced:5
Number produced:6
Number produced:7
Number produced:8
Number produced:9
...Not what we want
```

- The threads “trample” on the shared data
- The threads do not coordinate each other

Synchronization

- **Critical Section** – A method or a block of code managing a shared resource.
  - *Buffer.setNumber, Buffer.getNumber*
- **Synchronization - Mutual Exclusion** (Mutex) Enforcing limitations on accessing a critical section.
- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock.*
- Every object has a monitor lock associated with it.
  - A thread may *acquire, own, release* a lock

Synchronized

Preventing thread interference and memory consistency errors

- Synchronized Methods

```
public synchronized void setNumber(long number) {
    /* The producer acquires the monitor of the buffer object
    Throughout the execution of this method,
    the producer owns the buffer's monitor
    If the consumer invokes getNumber it will block
    (suspend execution) until the producer releases the lock */
    this.number = number;
    //The producer releases the monitor
}
public synchronized long getNumber() { … }
```

Warning
Thread Deadlock

- Synchronized Statements

```
public void setNumber(long number) {
    //Thread safe code - not accessing the buffer

    synchronized(this) {
        this.number = number;
    }
    //Thread safe code - not accessing the buffer
}
```

← specify whose lock are we using

Guarded Blocks

Some threads have to coordinate their actions

Semaphores

```
public class Buffer {
  private long number = -1;
  private boolean available = false;

  public synchronized long getNumber() {
    while (!available) {
      try {                                   ← Guarded Block
        wait();
      } catch (InterruptedException e) { e.printStackTrace(); }
    }
    available = false; notifyAll();
    return number;
  }

  public synchronized void setNumber(long number) {
    while (available) {
      try {                                   ← Guarded Block
        wait();
      } catch (InterruptedException e) { e.printStackTrace(); }
    }
    this.number = number; available = true; notifyAll();
  }
}
```

Wait – Notify

- ***Object.wait*** - **Causes the current thread to wait** until another thread invokes the *notify()* method or the *notifyAll()* method for this object. The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- ***Object.notifyAll*** - **Wakes up all threads that are waiting on this object's monitor.** The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

Atomic Access

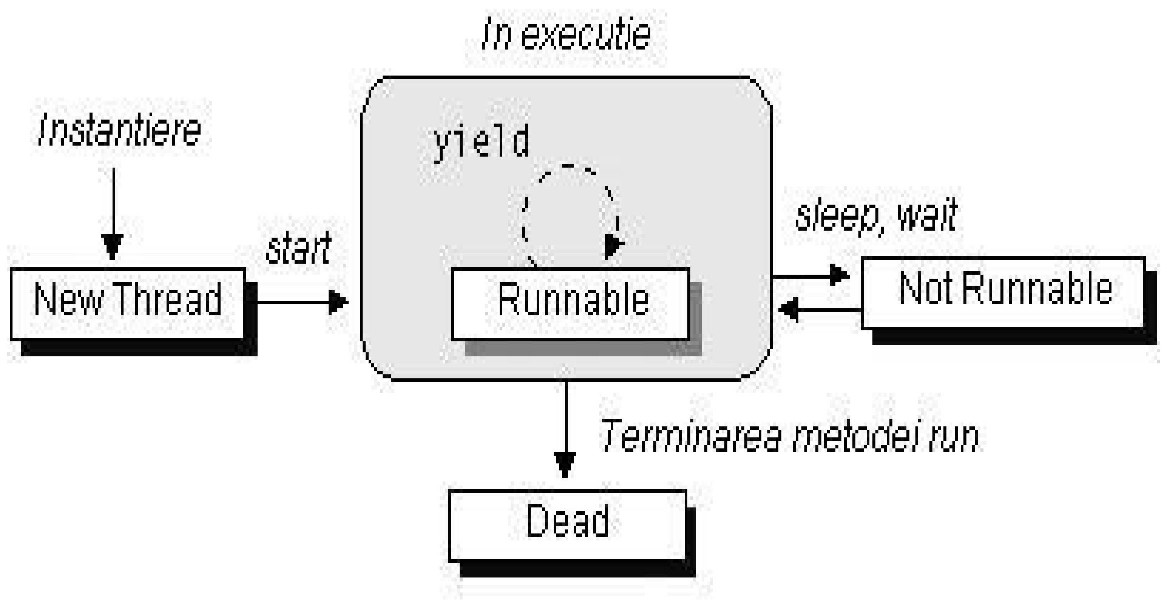
- **An atomic action cannot stop in the middle**: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.
- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared **volatile** (including long and double variables).
- **Each thread has its own stack**, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in **its own memory**.
- The Java **volatile** keyword is used to mark a variable as "being stored in main memory" → every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and every write to a volatile variable will be written to main memory, and not just to the CPU cache.

```
private volatile long number = -1;
```

Thread Scheduling

- Scheduling Models
  - **Co-operative** – time / **Pre-emptive** –resources
- The JVM is responsible with **sharing the available CPUs between all the runnable threads**. The JVM scheduler is usually dependent on the operating system.
- **Priority-based round-robin**
  - A thread of higher priority will preempt a thread of lower priority;
  - Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing.
  - Threads of equal priority will essentially take turns at getting an allocated slice of CPU;
  - Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY
- **Starvation and Fairness** → Watch out for **selfish** threads!

The Thread Lifecycle



- *sleep* - Causes the currently executing thread to temporarily cease execution.The thread does not lose ownership of any monitors.
- *yield* - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- *join* - Allows one thread to wait for the completion of another.
- *interrupt -* An interrupt is an indication to a thread that it should stop what it is doing and do something else → *InterruptedException*
- *~~stop~~, ~~suspend~~, ~~resume~~*

Stopping a Thread

- A thread dies naturally whenever its *run* method finishes its execution.
- ~~Thread.stop~~ – The thread is forced to stop whatever it is doing abnormally and to throw a newly created *ThreadDeath* object as an exception. Deprecated → This method is inherently unsafe...
- Killing it softly – Use a *control variable* to indicate that the target thread should stop running.

```
public class MyThread extends Thread {
    public boolean running = true;
    public void run() {
        while (running) {
            ...
        }
    }
}
```

```
MyThread t = new MyThread();
…
// Time to die.
t.setRunning(false);
```

Interrupting a Thread

It is not the same thing as stopping the thread

- *Blocking method* - takes a long time to run
  - involves invocation of *sleep(), wait(), join()*
  - should be *cancelable*
- Every thread → *interrupted status* property.

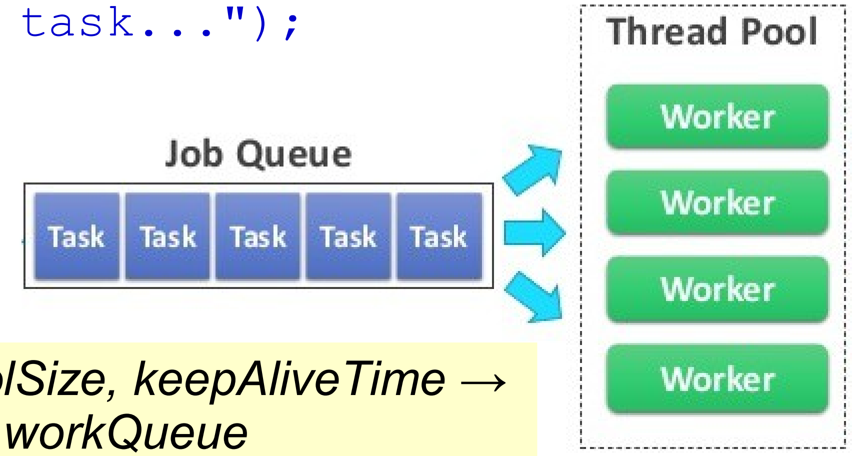
```
Thread t = new Thread() {
  public void run() {
    try {
      while(true) {
        //Perform some operation, wait 10 seconds
        Thread.sleep(10000);
      }
    } catch (InterruptedException e) {System.err.println(ex);}
    //Continue execution
    ...
  }
};
t.start();
...
t.interrupt(); // you should do something else
```

Thread Pool

Improving performance when executing large numbers of asynchronous tasks

- Allocating and deallocating many thread objects creates a significant memory management overhead.
- Instead of starting a new thread for every **task** to execute concurrently, the task can be passed to a **thread pool**.

```
ThreadPoolExecutor executor =
      (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
for (int i = 0; i <= 5; i++) {
   Runnable task = new Runnable() {
      public void run() {
         System.out.println("Doing a task...");
        }
   };
   executor.execute(task);
}
executor.shutdown();
```



corePoolSize, maximumPoolSize, keepAliveTime →
BlockingQueue<Runnable> workQueue

Fork/Join

Taking advantage of multiple processors

- Designed for work that can be broken into smaller pieces recursively and distributed to worker threads

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

- Uses a **work-stealing** algorithm.
  All threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks, as well as when many small tasks are submitted to the pool from external clients.

Using *ForkJoinPool*

```
public class ForkFind extends RecursiveAction {
    private final int[] numbers;
    private final int start, length, target;
    public ForkFind(int[] numbers, int start, int length, int target) {
        ...
    }
    protected void computeDirectly() {
      for (int i = start; i < start + length; i++) {
          if (numbers[i] == target)
              System.out.println("Found it at position: " + i);
      }
    }
    @Override
    protected void compute() {
        if (length < 1000) { computeDirectly(); return; }
        int split = length / 2;
        invokeAll(
          new ForkFind(numbers, start, split, target),
          new ForkFind(numbers, start + split, length - split, target));
    }
}
```

```
int numbers[] = new int[1_000_000]; int target = 1;
numbers[500_000] = target;
ForkFind ffind = new ForkFind(numbers, 0, numbers.length, target);
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(ffind);
```

ThreadLocal

- ThreadLocal variables that **can only be read and written by the same thread.** Even if two threads are executing the same code, the two threads cannot see each other's ThreadLocal variables.

```
public static class MyRunnable implements Runnable {
  private ThreadLocal threadLocal = new ThreadLocal();
  @Override
  public void run() {
    threadLocal.set( (int) (Math.random() * 100) );
    System.out.println(threadLocal.get());
  }
  public static void main(String args[]) {
    MyRunnable shared = new MyRunnable();
    Thread thread1 = new Thread(shared);
    Thread thread2 = new Thread(shared);
    thread1.start();
    thread2.start();
}
```

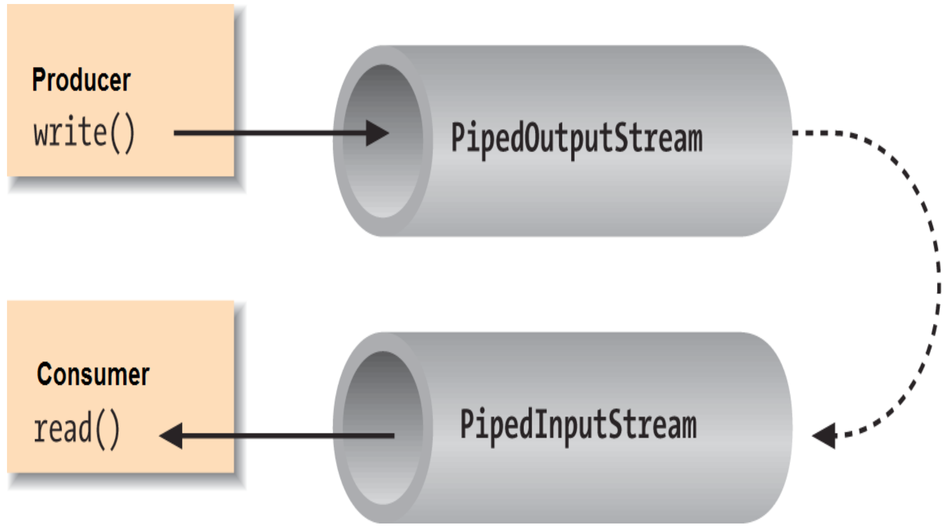
Each thread has its own, independently initialized copy of the variable.

Communication Pipes

```
class Producer extends Thread {
  private DataOutputStream out;
  public void run() {
    ...
    out.writeInt(i);
  }
}
class Consumer extends Thread {
  private DataInputStream in;
  public void run() {
    ...
    value = in.readInt();
  }
}
…
//A piped output stream can be connected to a piped input stream
PipedOutputStream pipeOut = new PipedOutputStream();
PipedInputStream pipeIn = new PipedInputStream(pipeOut);

DataOutputStream out = new DataOutputStream(pipeOut);
DataInputStream in = new DataInputStream(pipeIn);

new Producer(out).start();
new Consumer(in).start();
```



Concurrency in Swing

- JVM initially starts up with a single non-daemon thread, which typically calls the main method of some class.Once an application creates and displays a *Component* a new thread is created → **The Event Dispatch Thread**

```
Thread[] threads = new Thread[Thread.activeCount()];
Thread.enumerate(threads);
System.out.println(Arrays.toString(threads));
// → [Thread[main,5,main], Thread[AWT-EventQueue-0,6,main]]
```

- Swing **event handling and most code** that invokes Swing methods **run on this thread**, including the invocations to *paint* or *update*.
- When creating animations or complex drawings, time-consuming operations should be done in a separate thread and not in the EDT → Don't block the GUI
- Swing components should be accessed on the EDT only.

Don't Block The GUI

- When creating animations or drawing complex figures, time-consuming operations should be done in a separate thread.
- Wrong

```
public void paint(Graphics g) {
    // Complex calculations
    ...
    // Drawing
}
```

- Correct

```
public void paint(Graphics g) {
    // Drawing
}
```

```
//In another thread
public void run() {
  // Complex calculations
  component.repaint();
}
```

- Swing offers support for performing lengthy GUI-interaction tasks in a background thread *SwingWorker, SwingUtilities.invokeLater, ...*