Multitasking

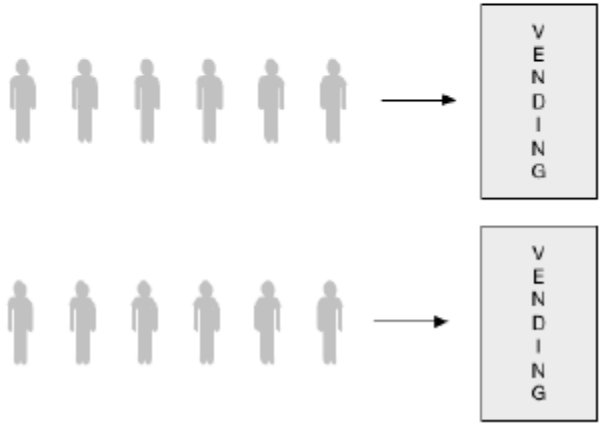
- Multitasking : the ability to have more than one program working at what seems like the same time.
  - Preemptive Multitasking (new OSs)
  - Cooperative (nonpreemptive) Multitasking (old OSs)
- Multithreaded programs extend the idea of multitasking by taking it one level lower: individual programs will appear to do multiple tasks at the same time.
- Each task is usually called a *thread of control*. A thread is an independent sequential path of execution within a program. Many threads can run concurrently within a program.

Multitasking (2)

- Concurrency and Parallelism

- A program is said to be concurrent if different parts of the program conceptually execute simultaneously
- A program is said to be parallel if different parts of the program physically execute simultaneously on distinct hardware
- A parallel program is concurrent but a concurrent program need not be parallel

Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines

Multitasking (3)

- The essential difference between multiple processes and multiple threads is that while each process has a complete set of its own variables, threads share the same data.
- Threads are more efficient and easier to program than inter-process communication. They are more "lightweight" than processes. Multi-threaded appplication = Application that supports concurrent execution of several threads. (concurrent threads).

Concurrency in Java

- Concurrent programming (multithreading) in Java. Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. Often the thread and its associated `Thread` object are thought of as being synonymous. The runtime environment distinguishes between *user threads* and *daemon threads.* Daemon threads exist only to serve user threads and stop when there are no more user threads running.
- When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the *main thread.*

Concurrency in Java

- All other threads, called *child threads*, are spawned from the main thread, inheriting its user-thread status. The `main()` method can finish, but the program will keep running until user threads (not daemons!) have completed
- Multi-threading in Java:
  - Who and how new threads are created and where the instruction sequence is located?
  - Thread life cycle;
  - How does a thread execute?

Thread Creation

- Best practices - using the newer, higher level java.util.concurrent library for managing concurrent programming in Java.
- Also necessary to understand the basic concepts first.
- Two ways of creating threads in Java:
- • creating a class that extends Thread and overriding run() method
- • creating a class that implements Runnable, overriding run() method and sending an instance of the class as a parameter to the constructor of a Thread.

- Example - implement a Counter that counts from one to three using both approaches. We create two counter threads and watch the output.
- Extending thread

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Main has started");
    CounterThread c1=new CounterThread("C1");
    CounterThread c2=new CounterThread("C2");
    c1.start();
    c2.start();
    System.out.println("Main has finished");
  }
}
```

```
public class CounterThread extends Thread{
  public CounterThread(String name) {
        super(name);
  }
  @Override
  public void run(){
    System.out.println("Thread "+getName()+ " has
started");
    for(int i=0;i<3;i++){
      System.out.println("Thread "+ getName()+" is at step
"+i);
      try {
        sleep(50);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("Thread "+ getName()+ " has
finished");
  }
}
```

- Observations:
  - from the output we see that even main is finished, the child threads continue to run and so does the application because they are user threads
  - method sleep() can throw an InterruptedException that we need to catch. This error is thrown if we call sleep() on a Thread in an Interrupted state
  - Output from the three threads is interlaced so this behaves like a multi-threaded application.
  - What if we change CounterThread to be a daemon??

Implementing Runnable

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Main has started");
    Runnable r = () -> {
      String name = Thread.currentThread().getName();
      System.out.println("Thread "+name+ " has started");
      for(int i=0;i<3;i++){
        System.out.println("Thread "+name+" is at step
"+i);
        try {
          Thread.sleep(50);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
      System.out.println("Thread "+name+ " has finished");
    };
    Thread c1 = new Thread(r);
    Thread c2 = new Thread(r);
    c1.setName("C1");
    c2.setName("C2");
    c1.start();
    c2.start();System.out.println("Main has finished");
```

- Observations
  - Runnable is a functional interface so we can replace the anonymous class definition with a lambda expression
  - In the interface we do not have an instance field so instead we access the Thread name by getting the currently executing thread and getting its name.
  - Similarly in order to call sleep method we call Thread.sleep() which calls sleep on the currently executing thread.
  - While this will become more evident in the java.util.concurrent library, we already see a clearer separation between the behavior specification (Runnable implementation) and runtime execution (Thread classes)

Thread's API

Costructors

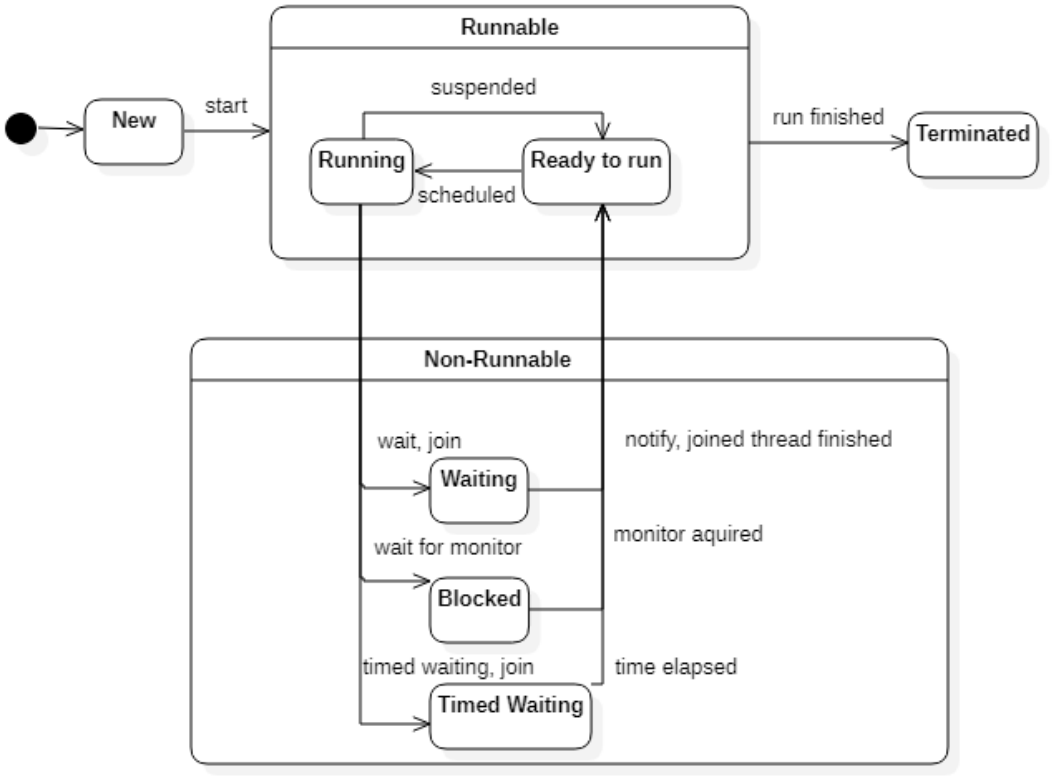
Thread()
Thread(Runnable target)
Thread(ThreadGroup group, Runnable target)
Thread(String name)
Thread(ThreadGroup group, String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name)

name – thread name. They are used to identify threads.
target – Runnable instance that is executed as the main method of the thread
group - the ThreadGroup name where the thread is added.

Methods

```
public final String getName();              // Returns this thread's name.
public final void setName(String name); // Changes the name of this thread to be equal to the argument name.
public final int getPriority();             // Returns this thread's priority.
public final void setPriority(int newPriority); // Changes the priority of this thread.
public final static int MAX_PRIORITY=10, MIN_PRIORITY=1, NORM_PRIORITY=5;
public final void start();                  // Causes this thread to begin execution; the Java Virtual Machine calls the run
                                               method of this thread.
public static void sleep(long millisec); //  Causes the currently executing thread to sleep (temporarily cease execution) for
                                               the specified number of milliseconds
public static void yield();                 // A hint to the scheduler that the current thread is willing to yield its current use of a
                                               processor.
public final void join();                   // Waits for this thread to die.
public final void join(long millisec);    // Waits at most millis milliseconds for this thread to die.
public void interrupt();                     // Interrupts this thread. If this thread is blocked in an invocation then its interrupt
                                               status will be cleared and it will receive an InterruptedException
public static boolean interrupted();       // Tests whether the current thread has been interrupted. The interrupted status of the
                                               thread is cleared by this method.
public boolean isInterrupted();             //  Tests whether this thread has been interrupted. The interrupted status of the thread
                                                is unaffected by this method.
public final ThreadGroup getThreadGroup(); // Returns the thread group to which this thread belongs.
public String toString();                   //  Returns a string representation of this thread, including the thread's name, priority,
                                               and thread group.
public static Thread currentThread();   // Returns a reference to the currently executing thread object.
public static int enumerate(Thread tarray[]);//Copies into the specified array every active thread in the current thread's
                                                 thread group and its subgroups.
public final boolean isAlive();             // Tests if this thread is alive. A thread is alive if it has been started and has not yet
                                               died.
public final boolean isDaemon();         //  Tests if this thread is a daemon thread (background test).
public final void setDaemon(boolean on); // Marks this thread as either a daemon thread or a user thread.
```

Thread Life Cycle



- *New :* thread has been *created*, but it has not yet *started*
- *Runnable :* thread which is either running or waiting for JVM to schedule it to run.
- *Terminated:* thread that has finished the execution of the run method or exited abnormally
- *Waiting:* thread that is waiting for another thread. A thread can enter in the waiting state if it calls wait on an Object or join on another thread.
- *Timed_waiting:* Similar to waiting, a thread that is waiting for another thread, waiting at most a specified amount of time
- *Blocked:* thread that is blocked while waiting for obtaining the lock to a monitor. A monitor is a mechanism that guarantees only one thread at a time can execute critical regions of code.

Stopping a thread

- Methods to directly stop the execution of a thread have become deprecated as they could leave the data the thread was modifying in an incoherent state.
- Instead, the accepted method is to set a flag, a boolean variable, in the thread, when we want it to stop. The thread can continuously check the flag and stop in an ordered way.

```
class StoppableThread extends Thread{
     boolean stop = false;
     public void run(){
       while (stop!=true){
         System.out.println("Thread is at step 1");
         System.out.println("Thread is at step 2");
         System.out.println("Thread is at step 3");
       }
     }

     public void setStop(){
       System.out.println("Stop set to true");
       stop=true;}
   }
```

interrupt() method

Instead of implementing your own original solution, Thread class already has a flag and method that accomplish what we implemented. Calling interrupt on a thread sets its interrupt flag to true, which can be checked with isInterrupted method. In addition, calling interrupt on a blocked thread will throw an InterruptedException, which wakes the thread and makes it deal with the exception.

The static *Thread.currentThread* method gets the current thread. We can obtain information if the interrupted status was set with *isInterrupted():*

```
while (!Thread.currentThread().isInterrupted() && more work to do) {
   do more work
}
```

```
Thread t = new Thread(){
      public void run(){
        while (isInterrupted() == false){
          System.out.println("Thread is at step 1");
          System.out.println("Thread is at step 2");
          System.out.println("Thread is at step 3");
          try {
            sleep(500);
          } catch (InterruptedException e) {
            System.out.println("Interrupt while blocked");
            break;
          }
        }
      }
    };
```

The *interrupt()* method in a blocked thread

If a thread is blocked when it is called with `interupt()`, it cannot check the interrupted status, but the blocking call (such as `sleep` or `wait`) is terminated with an `InterruptedException` and the interrupted thread can decide how to react to the interruption. Some threads are so important that they should handle the exception and continue.

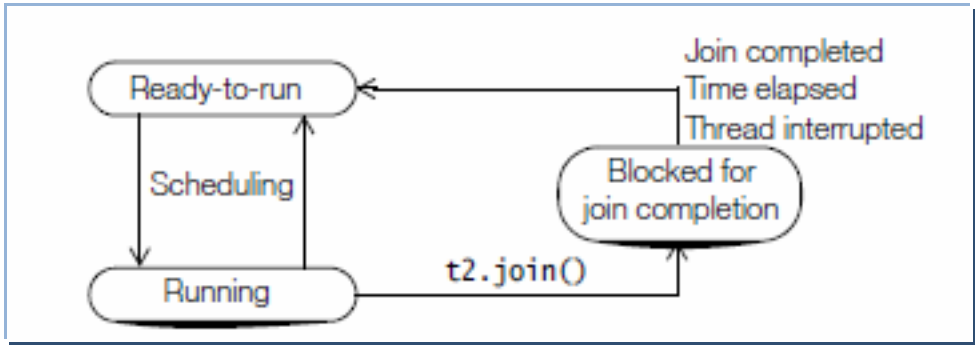
```
public void run() {
    try {
         . . .
       while(!Thread.currentThread().isInterrupted()&&more work to do){
            do more work
       }
    }     catch(InterruptedException e) {
          // thread was interrupted during sleep or wait
    }
      finally     {
          // cleanup, if required
    }
      // exiting the run method terminates the thread
   }
```

Thread Priorities

- Used by the thread scheduler
- Thread scheduler is platform-dependent
- 1-10, where 5 is the default priority
- Inherited from the parent thread
- `setPriority(int i)`

Thread joining

- A thread executing join() on another thread will wait the second thread to complete execution before continuing.
- It has no effect is the second thread has completed its execution



Thread synchronization

Since we have no control over how the scheduler gives control to one thread or another and considering that most operations in Java are not atomic (even what seems as a single operation as i++ is actually composed of more operations that can be interleaved) it is possible to have erroneous results if we allow multiple threads to write to the same location at the same time. We call a **critical section** a section of code that should be accessed by at most a thread at a time.

Synchronization fail

```
public class MainSync {
  public static void main(String[] args) throws
InterruptedException {
    Account a1=new Account("A1",10000);
    Account a2=new Account("A2",10000);
    Thread t1 =new Thread(() -> {
      for(int i=0;i<10000;i++){
        a1.deposit(1);
        a2.withdraw(1);
      }
    });
    Thread t2= new Thread(() -> {
      for(int i=0;i<10000;i++){
        a2.deposit(1);
        a1.withdraw(1);
      }
    });
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(a1);
    System.out.println(a2);
  }
}
```

Thread Synchronization

- Threads in Java have a basic way to synchronize access to critical sections of code – by using the monitor mechanism.
- A monitor consists of mutual exclusion based on locks and ability to cooperate using condition variables – waiting queues of threads that can be notified to be waked up when a condition becomes true.
- The simplest way to implement a monitor in Java is to add the synchronized keyword to methods or blocks in the objects shared by multiple threads.

Thread Synchronization

- Threads in Java have a basic way to synchronize access to critical sections of code – by using the monitor mechanism.
- A monitor consists of mutual exclusion based on locks and ability to cooperate using condition variables – waiting queues of threads that can be notified to be waked up when a condition becomes true.
- The simplest way to implement a monitor in Java is to add the synchronized keyword to methods or blocks in the objects shared by multiple threads.

Synchronization Rules

- Once a thread obtains it, no other thread can call any other synchronized methods on the same object until the first object releases the monitor lock – it finishes execution of the synchronized method.
- If synchronized is added to a static method, then the threads must obtain the class lock, meaning there can be only one thread at a time calling the synchronized static methods of a class. Object locks do not interfere with class locks.

Synchronization Rules

- Static methods synchronize on the class lock.
- Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.
- A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.
- A thread can hold a lock on an object
  - by executing a synchronized instance method of the object
  - by executing the body of a synchronized block that synchronizes on the object
  - by executing a synchronized static method of a class (in which case, the object is the Class object representing the class in the JVM)

Synchronized Blocks

- Overusing synchronized keyword can result in a degradation of performance – instead of having a multi-threaded program we now have a program in which only one thread is executing, and the others are waiting, negating the advantages offered by having more cores.
- A finer control can be given by using synchronized blocks with the syntax:

```
synchronized(object reference){
critical section
}
```

Synchronized Blocks

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
     synchronized (account) { // (1) synchronized block
       account.update(); // (2)
     }
  }
}
```

- The code at (2) in the synchronized block at (1) is synchronized on the BankAccount object. If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time only after synchronizing on the BankAccount object associated with this particular instance of SmartClient.

Solving synchronization

```
public class Account {
…

  public  synchronized void deposit(int sum){
    amount+=sum;
  }
  public  synchronized void withdraw(int sum){
    amount-=sum;
  }
}
```

wait ()

- The threads are usually independent:
  - They use shared memory for communicate with each other;
  - They execute identical operations on the same resources: file, images, databases.
- They collaborate with the methods *wait*() and *notify*() that enable them to interact with each other.

```
public synchronized void met() {
. . . .
      while (!condition) {
      . . . . .
        try{
              wait();
        } catch (InterruptedException e) {}
      } . . . .
}
```

- *wait*() is employed by a thread if it wants to leave the synchronized method waiting for an event.

```
public synchronized void withdraw(int amount) { //the thread access
    while(amount>balance)// to the object and nobody can enter the object
        wait();             // maybe another object deposit something
    balance -= amount;    // meantime in the current account and
}                                 // notifyAll() is called.
```

- Other wait methods:
  - wait(long *millisec*);
  - wait(long *millisec*, int *nanosec*);

notify()

- Calling wait() on an object puts the thread in the waiting list for that object. It will relinquish the lock on the current object but it will keep the lock on other objects. => possibility for deadlock
- Calling notify() on an object notifies the threads in the waiting list for that object. The selection of the thread who is woken up depends on the JVM. Woken thread must wait to reacquire lock
- notifyAll() wakes up all the threads in the waiting list
- wait(),notify(), notifyAll() are called on object whose lock the thread has so they are called in synchronized blocks or methods.

Producer consumer

- producer consumer problem states that we have two categories of threads – producers and consumers - sharing a bounded buffer or queue. Producers add items to the queue while consumers remove them. The producers and consumers need to synchronize to avoid the two unwanted situations that can occur: producers producing faster than consumers consume and adding more items than the queue capacity or consumers consuming faster than producers produce and trying to remove from an empty queue. This problem is found very often in distributed enterprise systems when the items can be requests or tasks.

Producer consumer

- The solution is straight forward: when the queue is full the producers wait until one item is removed and when the queue is empty the consumers wait until one item is produced.
- While there are dedicated data structures in Java already providing all the synchronization requirements let us implement from scratch a simple program where producers and consumers add and remove random integers to a Queue.

```
public class SharedRes {
  private Queue<Integer> queue= new LinkedList<>();
  private int limit;

  public SharedRes(int limit) {
    this.limit = limit;
  }
  public synchronized void produce(Integer i){
    while(queue.size() == limit) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
 notify();
    queue.add(i);
  }

  public synchronized Integer remove(){
    while(queue.isEmpty()) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    notify();
    return queue.remove(); }}
```

```
public class Producer extends Thread{
  private SharedRes sharedRes;
  public Producer(String name, SharedRes sharedRes) {
    super(name);
    this.sharedRes = sharedRes;
  }
  public void run(){
    Random ran = new Random();
    while(true){

      Integer i = ran.nextInt(100);
      sharedRes.produce(i);
      System.out.println(getName() +" added "+i);
    }  }}

public class Consumer extends Thread{
  private SharedRes sharedRes;

  public Consumer(String name, SharedRes sharedRes) {
    super(name);
    this.sharedRes = sharedRes;
  }

  public void run(){
    while(true){
      Integer i = sharedRes.remove();
      System.out.println(getName() +" removed "+i);
    }
  }
```

```
public class Main {
  public static void main(String[] args) {
    SharedRes shared = new SharedRes(5);
    Producer p1 =new Producer("P1",shared);
    Producer p2 =new Producer("P2",shared);
    Producer p3 =new Producer("P3",shared);
    Consumer c1 = new Consumer("C1",shared);
    Consumer c2 = new Consumer("C2",shared);
    Consumer c3 = new Consumer("C3",shared);
    p1.start(); p2.start(); p3.start(); c1.start();
c2.start(); c3.start();
  }
}
```