Concurrent Programming: multi-threading

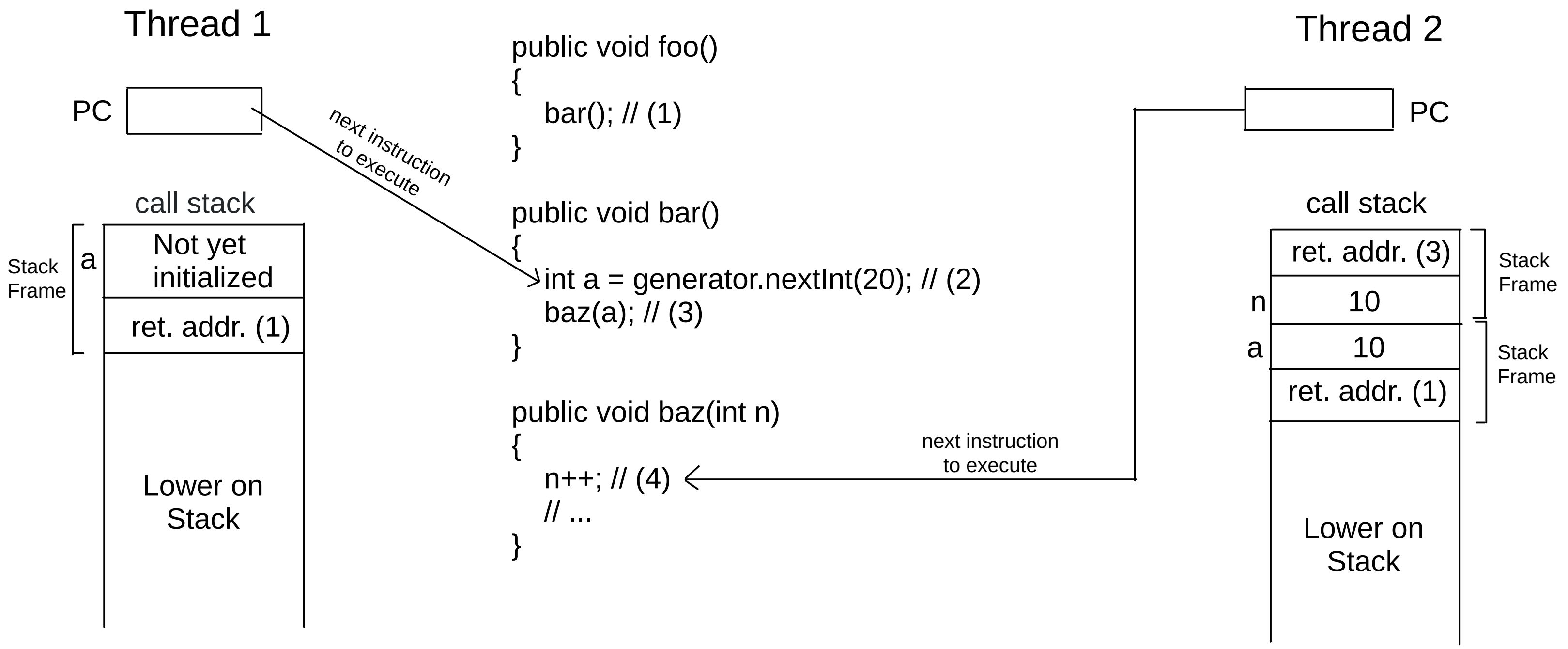
We are now going to do something cool. We are going to learn how to write a program that does multiple things at the same time (during overlaping time periods).

So far, the programs we have written only do one thing at a time. Java allows us to have multiple points of execution running at the same time in a single program. Each of these points of execution is called a thread.

A thread is a single sequential flow of execution withing a program.

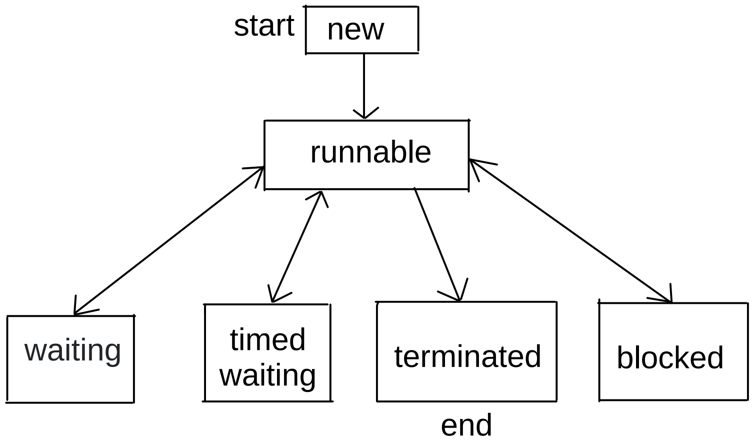
A program counter (PC) is a pointer (reference) to the current instruction to be executed by a thread.

Each thread has its own PC and call stack. The program heap is shared by all threads withing a single execution of a program. Example:



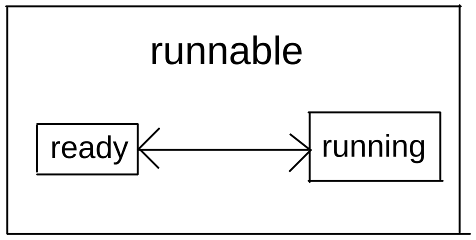
```
public void foo()
{
   bar(); // (1)
}

public void bar()
{
   int a = generator.nextInt(20); // (2)
   baz(a); // (3)
}

public void baz(int n)
{
   n++; // (4)
   // ...
}
```

At any given time a thread is in one of several states:



- A thread begins in the new state. It stays here until the program starts it, which places it in the runnable state.
- A thread in the runnable state is considered to be executing tasks.
- A thread transitions to the waiting state when it waits for another thread to perform its tasks.
- The timed waiting state is for threads that sleep (wait) for a specific interval of time.
- A thread moves to the blocked state when it attempts to perform a task that it cannot perform immediately.
- When a thread is done executing its tasks, it moves to the terminated state.

Typically the operating system splits the runnable state into two states called ready and running. These two states are hidden from the JVM.



- The ready state is for when a thread is ready to execute a task but is not actually doing so.
- A thread in the ready state transitions to the running state when it is assigned a processor and begins executing its tasks.

In Java, a class can allow its instances to run in their own thread by implementing interface Runnable (in java.lang).
Runnable has one method:

```
public void run()
```

which is where the tread begins to execute when started.

Example:

```
public class Nose implements Runnable
{
    // ...

    @Override
    public void run()
    {
        // Thread begins here when started
        // ...
    }
}
```

If your nose runs and your feed smell,
then you must be build upside down.

To create and start a new thread (from another one) that will begin at Nose's run() method, we can do:

```
Thread t = new Thread(new Nose()); // create a Thread object
t.start(); // Start tread t, which places it in the runnable state. It will start at the top of Nose's run() method.
           // The current thread that started t will not wait for t to finish executing. Instead, it will continue
           // with the next line of code right away. Both of these threads will be running in parallel at the same
           // time. The program terminates when all of its threads reach the terminated state.
```

(Continue on next page)

To put a thread to sleep (pause) for 1 second, we can write:

```
    try
    {
        Thread.sleep(1000);  // sleep for 1000 milliseconds
    }
    catch (InterruptedException e)
    {
        // catch clause: add code to execute if thread wakes up early. Can leave blank if there is nothing to do.
    }

    // We will talk about try-catch blocks more when we get to Exception.
```

This will put the current thread (the one executing the lines above) in the timed waiting state for 1000 milliseconds (1 second).

If the sleeping thread wakes up early (this could happen if the thread's interrupt() method is called), then it will run any code you put in the catch clause. Otherwise, when it wakes up it will skip the code in the catch clause.

Thread Pools and Java's Executor Framework

A thread pool is a collection of threads that are available for running tasks.

Thread pools can reduce the overhead of crreating new threads for each task by reusing available threads that already exist.

Java provides the Executor interface (in package java.util.concurrent) for managing the execution of Runnable objects. An Executor object typically creates and manages a thread pool.

The ExecutorService interface (in package java.util.concurrent) extends Executor and provides many additional useful methods.

To create a new ExecutorService object, we can do:

```
    ExecutorService es = Executors.newCachedThreadPool();  // Note: Executors here ends in an s
```

We can execute a Runnable object (in a thread) with our ExecutorService object es by doing:

```
    es.execute(new C());  // class C implements Runnable
```

Finally, when we are done submitting tasks to es, call:

```
    es.shutdown();
```

This will notify es to stop accepting new tasks (Runnable objects to execute), but will continue executing tasks that have already been submitted.

Thread synchronization

When more than one thread modifies shared data, then unexpected results may occur.

If one thread is in the process of updating some shared data and another thread tries to update it, it is unclear which thread's update takes place.

This can be fixed by allowing only one thread to access the shared data at a time - this is called mutual exclusion. Other threads must wait (are put in the blocked state) for the accessing thread to be done with the data. When the accessing thread is done, one of the waiting threads (if there are any) is given access to the shared data. This process is called thread synchronization.

Java has built in monitors for thread synchronization. Every object has a monitor and a monitor lock.

Mutual exclusion can be accomplished by giving an objects monitor lock to only one thread a time. To do this in Java, we can use a synchronized block:

Object whose monitor lock we are to use. This should be the object with the shared data.
This object is typically the invoking object (this).

```
    synchronized (obj)
    {
        // At most one thread is allowed to be running in here at any given time.
    }
```

1) When a thread t enters the above synchronized block, it will aquire the monitor lock (if available) from obj, preventing ALL other threads from running in ANY synchronized block on obj (often there are multiple synchronized blocks on an object).
2) If another thread tries to run in ANY synchronized block on obj (not necessarily the one above, but any synchronized block on obj), it will be placed in the blocked state and wait for t to release obj's monitor lock.
3) When t leaves the synchronized block, it will release obj's monitor lock, allowing another thread to aquire the the lock and run in a synchronized block on obj.

Note: obj should be the object with the shared data.

Example:
```
    synchronized (this)
    {
        this.sharedData += 2;  // suppose the invoking object has an int data field called sharedData
    }
```

Methods can also be synchronized
```
    public synchronized void write(int data)
    {
        // ...
    }
```

This is shorthand for:
```
    public void write(int data)
    {
        synchronized (this)
        {
            // ...
        }
    }
```

Note: A static method cannot be synchronized because it does not have an invoking object to synchronize on.
Static methods can have synchronized blocks in them though.

Sometimes a tread that enters a synchronized block cannot proceed until some condition is satisfied.

Example:

    B: Buffer of ints (shared data)
    P: Producer - thread that writes to B
    C: Consumer - thread that reads from B

    P and C run in separate threads.

    When C tries to read from B
    1) If B's monitor lock is available, then C aquires it. Otherwise C is blocked until the lock is available.
    2) If B is empty, then
        a) C releases the monitor lock so another thread can aquire it.
        b) C must wait for another thread to write to B. This is done by calling wait() <-- places C in waiting state
        c) When C wakes up:
            If B's monitor lock is available
            then C aquires the monitor lock
            else C is blocked until the lock is available.
        d) Go back to step 2 (to check if B is empty)
    3) Else B is not empty, so read from B.
    4) Notify all threads waiting on B to wake up. This is done with notifyAll()
    5) Release the monitor lock. This is done by exiting the synchronized block.

    Similar steps are done for when P tries to write to B.

Methods in class Object for writing and notifying other threads:

    public final void wait() throws InterruptedException
        - releases the monitor lock
        - puts the thread in the waiting state
        - throws an InterruptedException if another thread interrupts the current one

    public final void notifyAll()
        - Transitions all threads that are waiting for the invoking object's monitor lock to the runnable state. Each of these threads (including any threads blocked on the object) will try to aquire the lock. The rest of the threads will be blocked (put in blocked state).

(Methods in class Object for waiting and notifying other threads continued)

public final void notify()
- Like notifyAll() except only one thread waiting for the monitor lock is transitioned to the runnable state.
  This thread will attemp to aquire the monitor lock (blocked if it doesn't get the lock).

It is an error to call wait(), notifyAll(), or notify() on an object without having its monitor lock.

When to use notifyAll() and when to use notify()

Use notifyAll() when all waiting threads should eventually continue their task after the notification WITHOUT having to be notified again.

notifyAll() is usually what you'll use.

Use notify() when only one waiting thread (waiting on the invoking object) should continue its task after the notification. The other waiting threads (waiting on the invoking object) will each have to be notified again to continue their tasks (a call to notify() for each of the waiting threads).