# Java Threads (a review)

ICS432

**Concurrent and High-Performance Programming**

Henri Casanova (henric@hawaii.edu)

# Threads in Programming Languages

- Almost all programming languages provide constructs/abstractions for writing concurrent programs
  - even old ones (Modula, Ada, etc.)
- Java provides:
  - Thread class
  - Runnable interface
- Java also provides a Callable interface and higher level abstractions, which we'll see later in the semester
  - It's important to first master the "low-level" stuff

# Extending the Thread class

- Extend the thread class
- Override the **`run()`** method with what the thread should do
  - If you forget to override **`run()`**, your thread won't do anything
- Call the **`start()`** method to start the thread

**Thread subclass**

```
public class MyThread extends Thread {
  MyThread() { … }

  @override
  public void run() { // code for whatever the thread should do }
}
```

**Main program**

```
public class MyProgram {
  public static void main(…) {
     MyThread myThread= new MyThread();
     myThread.start();
     // At this point, 2 threads  are running!
  }
}
```

# run() vs. start()

- You **implement** the thread's code in **`run()`**
- You **start** the thread with **`start()`**

- **WARNING**: Calling **`run()`** does **not** create a thread, but it works (it's just a normal method call)
- The **`start()`** method, which you should not override, does all the thread launching
  - It places whatever system calls are needed to start a thread (e.g., the **`clone`**, aka **`fork`**, system call in Linux)
  - And then makes it so that the newly created thread's fetch-decode-execute cycle begins with the first line of code of the run() method

# The Runnable Interface

- Using the Runnable interface is preferred because then you can still extend another class
  - Java doesn't have multiple inheritance
  - Typically if you can use an **`implements`** instead of an **`extends`**, you should
    - So that you keep the **`extends`** option open for another purpose
- Let's see an example…

# Using the Runnable Interface

Runnable class

```
public class MyRunnable implements Runnable {
  MyRunnable() { … }

  @override
  public void run() { // code for whatever the thread should do }
}
```

Main program

```
public class MyProgram {

  public static void main(…) {
     //  Create an instance of the runnable class
     MyRunnable myRunnable = new MyRunnable();
     // Pass it to the Thread constructor
     Thread thread = new Thread(myRunnable);
     // Start the thread
     thread.start();
    // At this point, 2 threads  are running!
  }
}
```

# In-line Thread Creation

- Sometimes it's cumbersome to create all kinds of Runnable classes, so one can "lambda it" :)

Main program

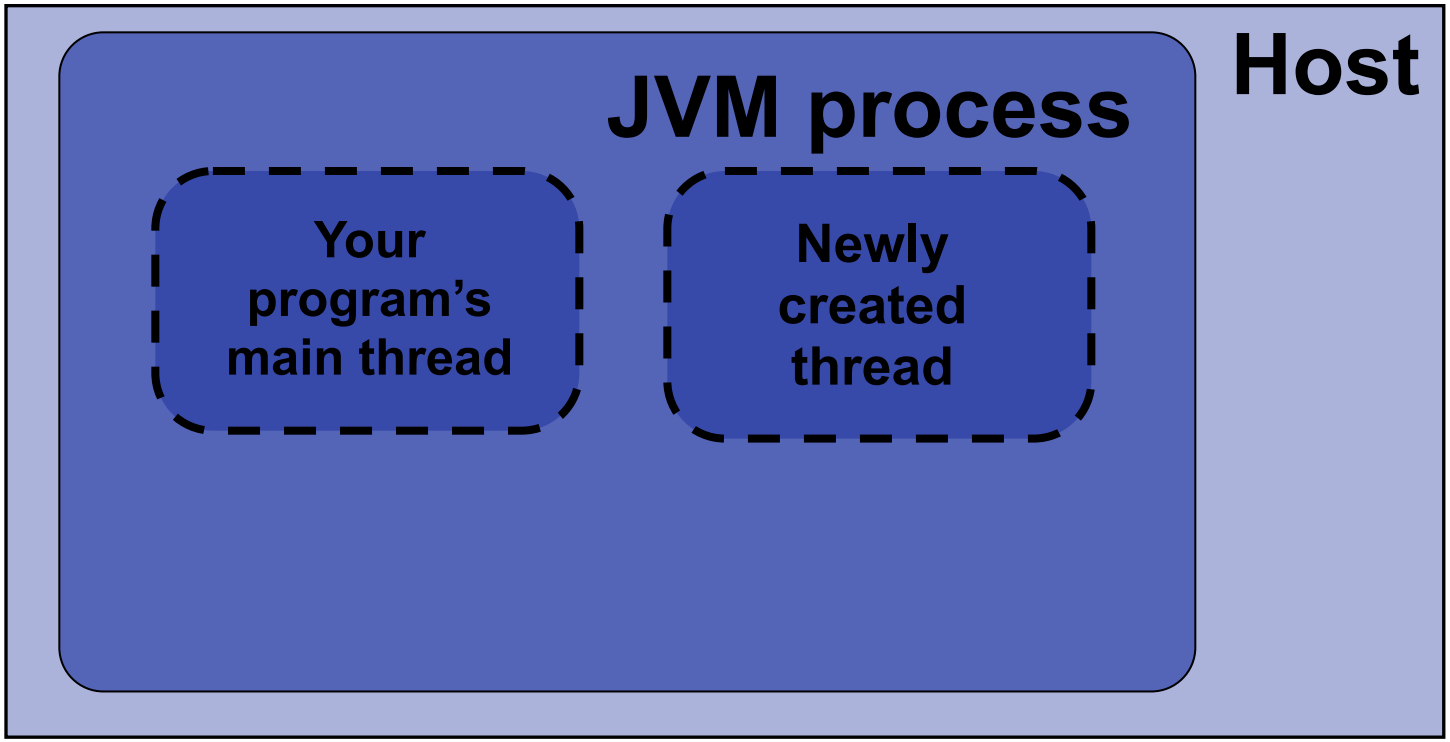
```
public class MyProgram {

  public static void main(…) {

    // Start an anonymous thread with a single statement
    new Thread( new Runnable() {
       @Override
       public void run() {
          …
       }
     }).start();

   }
}
```

# The isAlive() Method

- After you spawned a thread you may not really know if it's terminated or not
- It may be useful to know
  - To see if the thread's work is done for instance
- The **`isAlive()`** method returns true is the thread is running, false otherwise
- Could also be useful to remember whether you have called **`start()`** on a thread, or to restart a thread

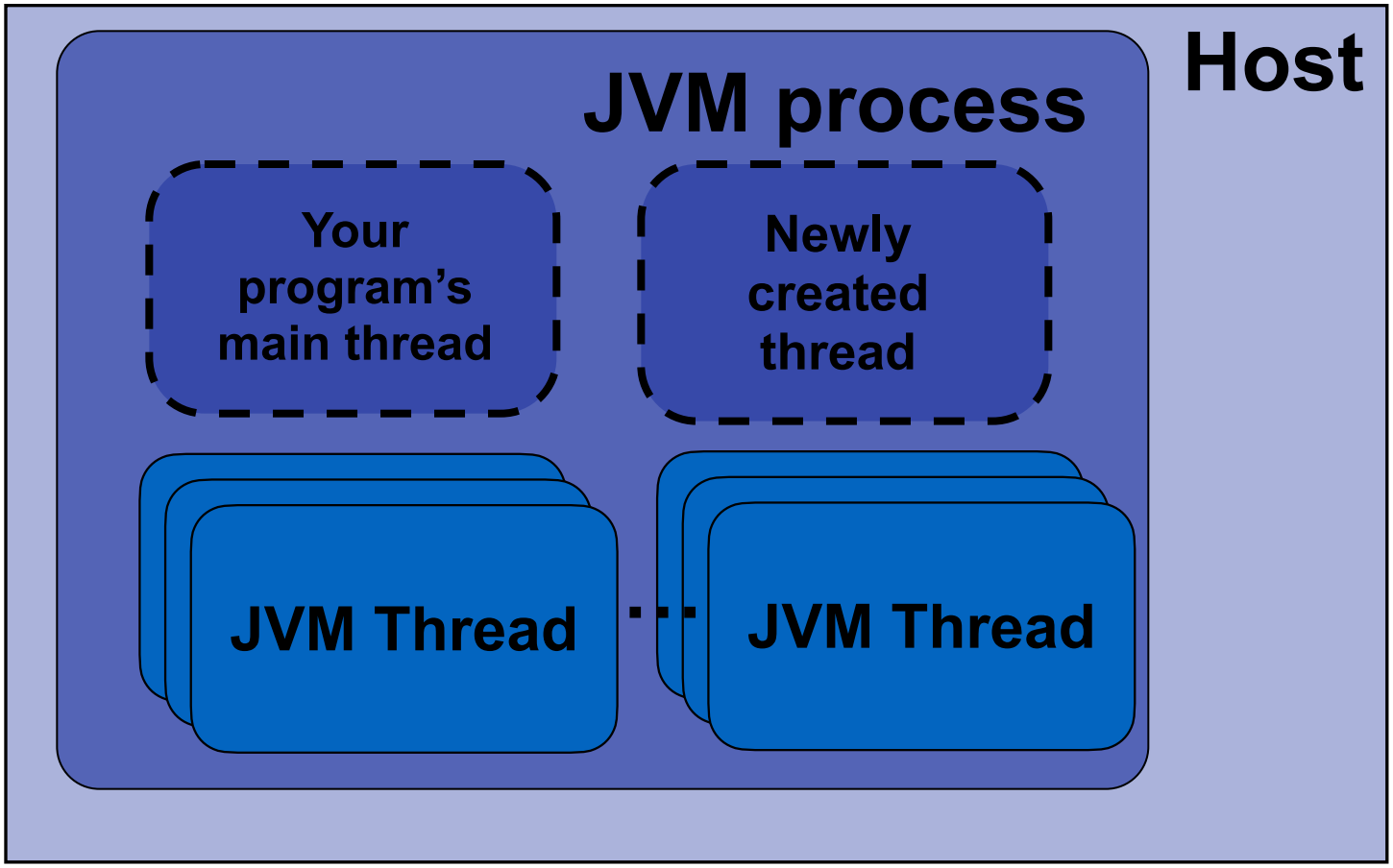
```
if (!t.isAlive()) {
  t.start();
}
```

# Your Two Threads




- A Java program terminates only when all **your** threads have terminated (unlike in many other languages)
- But there are many more threads in the JVM!
  - Let's find out how many by writing some code…

# Many Threads in




- Thread JVM threads are called "daemon threads"
- Some you already know about (garbage collector), some we'll talk about (JavaFX Application Thread), some we won't discuss

# Non-deterministic Execution!

- Remember from your OS course that the OS schedules when a thread runs (by taking it out of the ready queue and giving a time quantum on one core)
  - In ICS332 you learned some of the “smarts” implemented in the kernel to schedule threads efficiently
- So if your main thread prints a bunch of “*” and your newly created thread prints a bunch of “#”, there is no way to tell what the output will be
  - And the output will be different each time
- This can make debugging really difficult
  - The age-old “my program breaks only once every 1000 executions”
- But you cannot make assumptions about thread scheduling since  the OS is in charge, not you

# Influencing Thread Scheduling

- We throw a bunch of threads in, the OS "shakes the bag", and we don't really know what will happen
- But the JVM provides some ways to influence what happens
  - Thread.yield()  (a static method)
  - Thread.setPriority(int p)  (a non static method)
- Let's review these briefly…

# Thread.yield()

- When a thread calls yield() it is saying "I am willingly giving up the CPU right now”
- Somehow, many programmers use yield(), typically to ensure some interactivity, sprinkling their code with yield() calls everywhere
- Since Java 7, the Javadoc has been saying ***“A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint [ …] It is rarely appropriate to use this method.”***
- **DO NOT USE Thread.yield()**
  - Only acceptable use: for debugging purposes

# Thread Priorities

- The Thread class has a setPriority() and a getPriority() method
  - A new Thread inherits the priority of the thread that created it
- Thread priorities are integers ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY
  - The higher the integer, the higher the priority

# Thread Priorities and Scheduling

- Whenever there is a choice between multiple runnable threads, the JVM should pick the higher priority one
- The JVM is preemptive
  - If a new higher priority thread is started, it gets to run now
- In spite of all this:
  - The JVM can only *influence* the way in which threads are scheduled
  - Ultimately, the decision is left to the OS
- So, again, these are hints: A JVM is free to implement priorities in any way it chooses, including ignoring the value!
- A few years ago I had designed a programming assignment that used priorities, and half the students in the class had a JVM implementation that ignored priorities!

# Influencing Thread Scheduling?

- The Java API provides a few methods for this, as we saw, but they just cannot be relied upon for correctness
  - After all, the JVM is not the OS, so it's not in charge
- So  if you use these methods, your program may work ok on  your JVM and your machine, but not ok at all on another system
- Bottom Line: We have to rely on other (deterministic) mechanisms to orchestrate the execution of our threads
- Let's see the simplest such mechanism…

# The join() method

- The `join()` method causes a thread to wait for another thread's termination

Example program

```
public class JoinExample {
  public static void main(String args[]) {
    // Create a thread
    Thread t = new Thread (new Runnable() {
      @Override
      public void run() { . . . }});

    // Spawn it
    t.start();

    // Do some work myself
    . . .

    // Wait for the thread to finish
    try {
      t.join();
    } catch (InterruptedException ignore) {}
  }
}
```

# The setDaemon() method

- Sometimes we want to start threads that will run forever, but we want them to be “daemon threads”
  - i.e., the program can terminate even though these threads are still running
  - Remember that by default a Java program does not terminate until all its non-daemon-threads have terminated

```
Thread t =  new Thread(…);
t.setDaemon(true);
```

# Conclusion

- Best way to create threads in Java: implement the Runnable interface and create a new Thread object
- Thread scheduling is complex, not deterministic, and providing hints to the JVM must not be relied upon to guarantee program correctness
- Up next: super-quick JavaFX intro