Threads

Motivation

- an event handler that runs for 5 seconds freezes the whole window
  - the JavaFX Application Thread is busy running the handler so it can't repaint or handle other mouse and key events
- sometimes a program needs to do more than one thing at once
  - a GUI: carry out a long job while still handling repaints and new events
  - a server: wait for new clients while serving the ones already connected
  - blocking operations: keep doing other work while one part of the program waits
    - on a server: `accept()` blocks until a client connects
    - I/O: `read()` blocks until there's something to read
  - speed: finish a big job sooner by splitting it across several cores
  - the system: perform background work such as garbage collection

Concurrency vs Parallelism

- *concurrency* – several tasks in progress at once
  - they may actually run at the same time on different cores, or only appear to with the OS interleaving their steps on one core
- *parallelism* – several tasks actually running at the same time on different cores
  - parallelism is one kind of concurrency, not an alternative to it
- in Java, *threads* are the mechanism for both
  - the program creates threads
  - the system decides whether they really run in parallel

Threads

- a *thread* is one path of execution through a program
  - most programs have a single thread of control
    - one finger is enough to keep track of where you are (the current line)
  - a *multithreaded* program has several threads at once
    - you need a finger for each to keep track of multiple current lines

Threads

- each thread has its own *call stack* (local variables, parameters) and current position in the code
- all threads share the *heap* (containing every object created with `new`) and static fields (one copy per class)
  - multiple threads can access the same objects – this is both a convenience and a risk
  - whether a thread can access something depends on whether it can reach it (through a reference or by name)
  - access modifiers like `private` control which code can name a field, not which threads can use it
- the scheduler decides which thread runs, when, and for how long
  - your code can't predict or assume a particular order without explicit coordination

Threads in Java

- Java programs are inherently multithreaded *
  - one main thread, which runs `main` and everything called from it
  - zero or more programmer-created threads
  - several system threads: the JavaFX Application Thread (painting, event handling), garbage collection
- command-line programs exit when all of the non-system threads have finished, not when `main` returns
- GUI programs exit when the last window has been closed or `Platform.exit()` has been called

* though the term *multithreaded* is generally reserved for when there are programmer-created threads

Questions

Is there a maximum number of threads a program can have?

- yes, but the particular number depends on…
  - available memory (directly on stack size, indirectly on heap size)
    - can change the defaults for the JVM, but ultimately limited by system's memory
  - the underlying operating system
    - macOS and Linux limit the number of threads
    - Windows does not

Creating Threads

- three steps
  - define the code for the thread to run
  - create the thread object
  - make the thread ready to run

Creating Threads

- define the code for the thread to run
  - class extends `Runnable`
  - code for the thread to run goes into `run()`

```
public class Countdown implements Runnable {

        private int value_;
        private String indent_;

        public Countdown ( int value, String indent ) {
                value_ = value;
                indent_ = indent;
        }

        @Override
        public void run () {
                System.out.println(indent_+Thread.currentThread().getName());
                for ( ; value_ >= 0 ; value_-- ) {
                        System.out.println(indent_ + value_);
                }
                System.out.println(indent_ + "done!");
        }
}
```


Creating Threads

- create the thread object
- make the thread ready to run
  - `start()` does the internal setup to create the thread and returns
  - the scheduler calls `run()` – your code never calls run() directly

```
Thread thread1 = new Thread(new Countdown(20,""));
Thread thread2 = new Thread(new Countdown(20,"   "));

thread1.start();
thread2.start();
```


Creating Threads – Alternatives

- with a lambda
  - `Runnable` only has one method, so a lambda can stand in for it

```
Thread thread3 = new Thread(() -> {
        for ( int i = 3 ; i >= 0 ; i-- ) {
                System.out.println(i);
        }
});

thread3.start();
```

- by subclassing `Thread` and overriding `run()`

```
public class CountdownThread extends Thread {
        ...                             // same fields and constructor as Countdown
        @Override
        public void run () { ... }      // same body as Countdown.run()
}

Thread thread4 = new CountdownThread(20,"      ");
thread4.start();
```

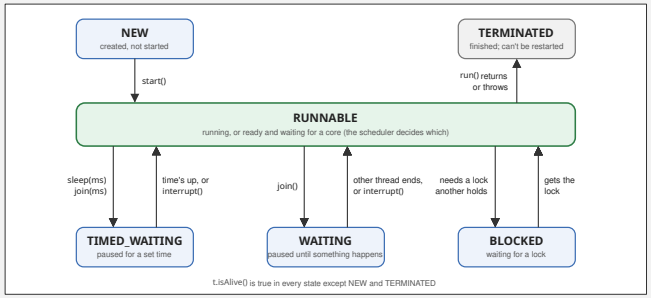

Creating Threads – Alternatives

- implementing `Runnable` is the preferred way
  - "this code can be run by a thread"
  - separates what runs from how it is run
    - `run()` can be called directly (without creating a new thread) e.g. for testing
    - the task can be handed to a thread pool, which runs it on an existing worker thread
- a lambda saves writing a separate class
  - it can't take constructor arguments but can use local variables from where it is written if they don't change afterwards (effectively final)
  - it can't have fields or other methods, such as `shutdown()` or `getCount()`
- extend `Thread` only if you need to override other methods of `Thread`
  - "this code is a thread"
  - can't extend other classes

Demo

- basic countdown

Thread Lifecycle



- a thread is in exactly one state at a time
- key points
  - `t.isAlive()` is true for every state other than `NEW` and `TERMINATED`
  - `start()` only works once – moves thread from `NEW` to `RUNNABLE`
  - `interrupt()` wakes a thread that is sleeping or joining, but not one that is blocked
  - a `TERMINATED` thread can't be restarted
  - join loop checks `isAlive()` to determine if thread has terminated or has only been interrupted

Thread Management Patterns

- pausing and periodic tasks
- waiting for a thread to finish
- stopping a thread
- ending a program

Pausing and Periodic Tasks

```
public void run () {
        for ( ; value_ >= 0 ; value_-- ) {
                System.out.println(indent_ + value_);
                try {
                        Thread.sleep(1000);
                } catch ( InterruptedException e ) {}
        }
        System.out.println(indent_ + "done!");
}
```

- `Thread.sleep(ms)` pauses the *current* thread for *about* `ms` milliseconds
  - this is a `static` method so always pauses the thread that calls it
    - `thread1.sleep(500)` compiles (legal to call a static method on an object) but makes the caller sleep, not thread1
  - there are ways to wake up a sleeping thread early, which results in an `InterruptedException`
- this is the basic pattern for a timer
  - `run()` has a loop whose body does the task and then sleeps
  - where the `try`/`catch` goes depends on what an interrupt does – skip this round (task and pause), carry on (skip only the rest of the pause), or end the thread

Demo

- timer demo (slow countdown)

Waiting For a Thread to Finish

```
System.out.println("waiting for " + thread1.getName() + " to finish");
for ( ; thread1.isAlive() ; ) {
        try {
                thread1.join();
        } catch ( InterruptedException e ) {
                System.out.println("I was interrupted!");
        }
}
System.out.println("waiting for " + thread2.getName() + " to finish");
for ( ; thread2.isAlive() ; ) { ... }

System.out.println("all threads done");
```

- `t.join()` waits until `t` has terminated
  - it returns immediately if `t` has already terminated
- `join()` can be interrupted so put it in a loop to wait until the thread has actually terminated
- only wait for a thread if something must happen only after it terminates e.g. cleanup or using its result

Demo

- join demo

Stopping Threads

- `Thread` methods `stop()` and `destroy()` have been deprecated
  - `stop()` is unsafe
    - stopping a thread unlocks its locks, meaning mutual exclusion may no longer be ensured and inconsistent state may result
  - `destroy()` may result in deadlock
    - the destroyed thread doesn't release its locks, so any thread trying to acquire the same locks is stuck



- instead, threads must be designed to be stopped safely
  - the "status flag" approach – the task has a `shutdown()` method that sets a flag, `run()` checks the flag regularly and ends itself

The Status Flag

status flag – set to true when shutdown is requested, must be monitored in `run()`

another thread calls `shutdown()` to request a shutdown

```
public class SlowCountdown2 implements Runnable {

  private int value_;
  private String indent_;

  private volatile boolean shutdown_;
  private volatile Thread thread_;

  public SlowCountdown2 ( int value, String indent ) {
    value_ = value;
    indent_ = indent;
    shutdown_ = false;
    thread_ = null;
  }

   * (non-Javadoc)
  @Override
  public void run () {
    thread_ = Thread.currentThread();
    for ( ; value_ >= 0 && !shutdown_ ; value_-- ) {
      System.out.println(indent_ + value_);
      try {
        Thread.sleep(1000);
      } catch ( InterruptedException e ) {
        System.out.println(indent_ + "interrupted!");
        if ( shutdown_ ) { break; }
      }
    }
    System.out.println(indent_ + "done!");
  }

  public void shutdown () {
    shutdown_ = true;
    thread_.interrupt();
  }

}
```

```
System.out.println("shutting down " + thread2.getName());
countdown2.shutdown();
```


The Status Flag

- a wrinkle is that if the thread is blocked on something, it won't notice that `shutdown_` has been set until it wakes up
  - solution is application-specific but involves waking the thread up from whatever it is blocked on
    - e.g. interrupt if sleeping, close stream if reading

`sleep()` might take a while

`interrupt()` interrupts `sleep()` – needs to be invoked on the thread to be shut down, so that must be stored (remember that the thread calling `shutdown()` is not the same thread as the one being shut down or the one calling the constructor – `thread_` must be initialized in `run()`)

```
public class SlowCountdown2 implements Runnable {

  private int value_;
  private String indent_;

  private volatile boolean shutdown_;
  private volatile Thread thread_;

  public SlowCountdown2 ( int value, String indent ) {
    value_ = value;
    indent_ = indent;
    shutdown_ = false;
    thread_ = null;
  }

  /* (non-Javadoc)
  @Override
  public void run () {
    thread_ = Thread.currentThread();
    for ( ; value_ >= 0 && !shutdown_ ; value_-- ) {
      System.out.println(indent_ + value_);
      try {
        Thread.sleep(1000);
      } catch ( InterruptedException e ) {
        System.out.println(indent_ + "interrupted!");
        if ( shutdown_ ) { break; }
      }
    }
    System.out.println(indent_ + "done!");
  }

  public void shutdown () {
    shutdown_ = true;
    thread_.interrupt();
  }

}
```

interrupted – check if it was because thread should shut down

```
System.out.println("shutting down " + thread2.getName());
countdown2.shutdown();
```


The Status Flag

one thread calls `shutdown()` to set `shutdown_`, another thread reads `shutdown_` in `run()`

without `volatile`, the reader might never see the change – next lecture explains why

```
public class SlowCountdown2 implements Runnable {

  private int value_;
  private String indent_;

  private volatile boolean shutdown_;
  private volatile Thread thread_;

  public SlowCountdown2 ( int value, String indent ) {
    value_ = value;
    indent_ = indent;
    shutdown_ = false;
    thread_ = null;
  }

   * (non-Javadoc)
  @Override
  public void run () {
    thread_ = Thread.currentThread();
    for ( ; value_ >= 0 && !shutdown_ ; value_-- ) {
      System.out.println(indent_ + value_);
      try {
        Thread.sleep(1000);
      } catch ( InterruptedException e ) {
        System.out.println(indent_ + "interrupted!");
        if ( shutdown_ ) { break; }
      }
    }
    System.out.println(indent_ + "done!");
  }

  public void shutdown () {
    shutdown_ = true;
    thread_.interrupt();
  }

}
```

```
System.out.println("shutting down " + thread2.getName());
countdown2.shutdown();
```


Demo

- shutdown demo

Ending a Program

- ending a program
  - a command-line program doesn't exit until all non-system threads have finished
    - *daemon* threads (`t.setDaemon(true)`, called before `start()`) doesn't keep the program alive
  - `System.exit()` ends everything at once – risky
    - better practice is to shut down each thread that won't finish on its own so it can complete its work and release resources
  - a JavaFX program exits when the last window closes or `Platform.exit()` is called
    - override `Application.stop()` if your threads need shutting down then

Notes

- name your threads – `new Thread(countdown,"countdown")`
  - default is Thread-0, Thread-1, …
  - names show up in stack traces, logs, the debugger and can be printed for your own debugging
- an uncaught exception ends only its own thread
  - other threads (including the `main` thread) aren't told – they just continue on