Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Threads

Running a Java Thread

Two ways to run a Java thread:

- **Runnable** - Write a class that implements the Runnable interface and pass an instance of it to a Thread instance.

OR

- **Derive Class From Thread** - Write a class that inherits from the Thread class.

```
public class RunThreadImplementRunnable implements Runnable {

   @Override
   public void run() {
      System.out.println("Thread - RunThreadImplementRunnable");
   }
}

public class Driver {
   public static void main(String[] args) {
      System.out.println("Thread - Main Started");

      Thread t = new Thread(new RunThreadImplementRunnable());
      t.start();

      System.out.println("Thread - Main Finished");
   }
}
```

Override Runnable's run method

Class implements the Runnable Interface

Create an instance of the class that implements Runnable and pass it to the Thread constructor

Call start to actually run the thread

Implementing Runnable

```
public class RunThreadExtendThread extends Thread {
   @Override
   public void run() {
      System.out.println("Thread - RunThreadExtendThread");
   }
}

public class Driver {
   public static void main(String[] args) {
      System.out.println("Thread - Main Started");

      RunThreadExtendThread threadExtend = new RunThreadExtendThread();
      threadExtend.start();

      System.out.println("Thread - Main Finished");
   }
}
```

Class inherits from Thread

Override the run method

Create instance of the Thread derived class

Call start to actually run in a thread

Extending Thread Class

- **Anonymous Runnable** - The following code runs a thread using an anonymous instance of Runnable:

```
public static void main(String[] args) {
   Thread t = new Thread(new Runnable() {
      @Override
      public void run() {
         System.out.println("Thread - I love Java!");
      }
   });
   t.start();
}
```

The code in blue defines the anonymous Runnable instance.

The thread constructor parameter is within the red parenthesis. An anonymous instance of Runnable is created and passed into the Thread constructor.

Anonymous Runnable

Anonymous Runnable Using Lambda

- **Lambda for Anonymous Runnable** - The following code runs a lambda expression to define the anonymous instance of Runnable:

The lambda expression for the anonymous Runnable is in blue. Runnable is a functional interface (interface has only one method) so we can use a lambda expression to define it.

```
public static void main(String[] args) {

   Thread t = new Thread( () -> {
      System.out.println("Thread - I love Java with lambdas!");
   });
   t.start();

}
```

The thread constructor parameter is within the red parenthesis

- **Pass Data to Thread** - You may need to pass data to a thread.
- Create a Runnable class having member variables to hold the parameters.
- Pass the parameters in using a constructor.
- The run method will have access to the member variables.

```
public class MyRunnableWithParameter implements Runnable {
    private int data;

    public MyRunnableWithParameter(int d) { data = d; }

    @Override
    public void run() {
       System.out.printf("Passed in data is: %d\n", data);
    }
}
Thread t = new Thread( new MyRunnableWithParameter(777) );
t.start();
```

Pass the data in using the constructor

run has access to the member variables which contain the data

Create instance of MyRunnableWithParameter in main using the constructor to pass in data

Pass Data to Thread

- **Pass Data to Thread Using final Variable**

```
public static void main(String[] args) {
   final String message = "I love threads with final variables!";

   Thread t = new Thread( () -> {
      System.out.println(message);
   });
   t.start();
}
```

The message variable is declared as final in the surrounding method so it can be used in the thread

Pass Data to Thread Using final Local Variables

Join – Wait for another thread to finish

- **join** - If you need to wait for a thread to finish use the Thread class join method.
- When join is called the calling thread must wait for the thread instance that join is called on to finish before it can proceed.

```
public static void main(String[] args) {
   Thread t = new Thread(new RunThreadImplementRunnable());
   t.start();

   try {
      t.join();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

Sleep

- **sleep** – Stops the current executing thread for the given amount of time (in milliseconds).

```
public static void main(String[] args) {
   Thread t = new Thread( () -> {
      System.out.println("Thread – Going to sleep");
      Thread.sleep(1000);
      System.out.println("Thread – Waking up!");
   });
   t.start();
}
```

Thread t will sleep for 1 second (1000 milliseconds)

Synchronized Method

- **Synchronized Method** – You can decorate a method with the synchronized keyword to create a critical section.
- Only one thread will be allowed to be running this method at any moment in time.
- For example, if thread A is running myMethod and thread B calls myMethod then thread B will have to wait until thread A finishes running it.

Decorate a method with the synchronized keyword to only allow one thread at a time to run it

```
public synchronized void myMethod()
{
   // code for critical section goes here…
}
```

- **Synchronized Block**- You can use a synchronized block to create a critical section.
- You must create an instance of Object that will serve as the lock for the synchronized block.

```
Object myLock = new Object();
```

The lock object instance must be accessible to all threads that will use it

```
public void myMethod()
{
   // Other code not in critical section can go here…

   synchronized(myLock)
   {
      // code for critical section goes here…
   }
 // Other code not in critical section can go here…
}
```

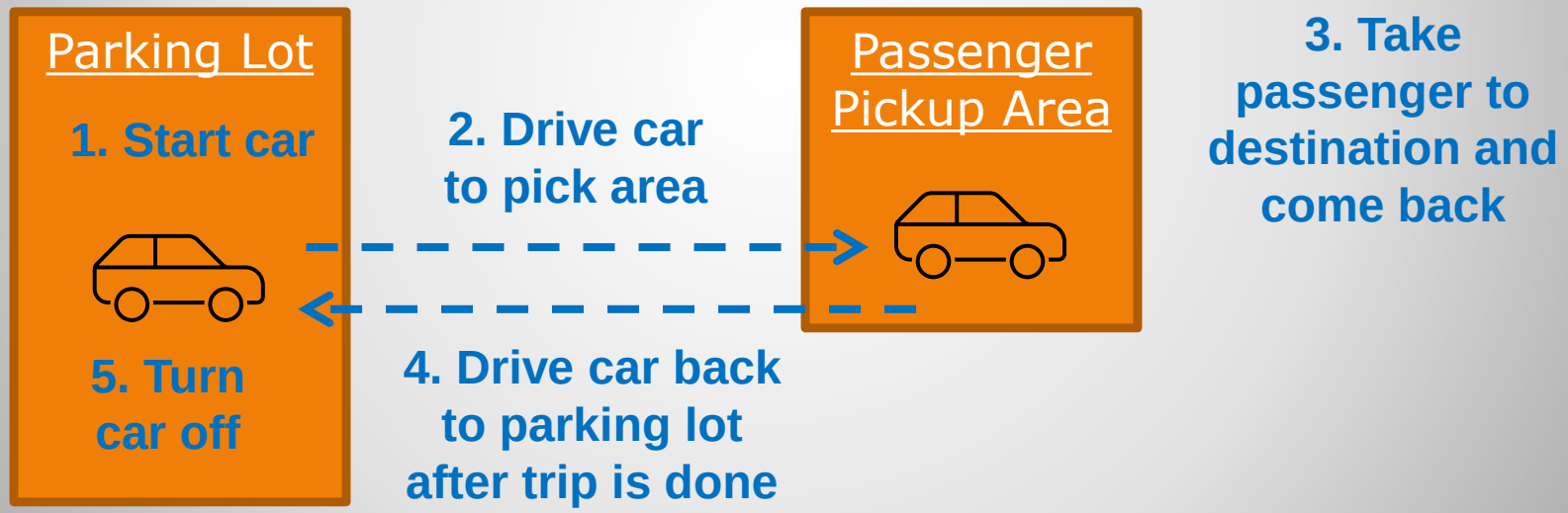
Use the lock here

Synchronized Block

- Now on to higher-level ways to use threads….

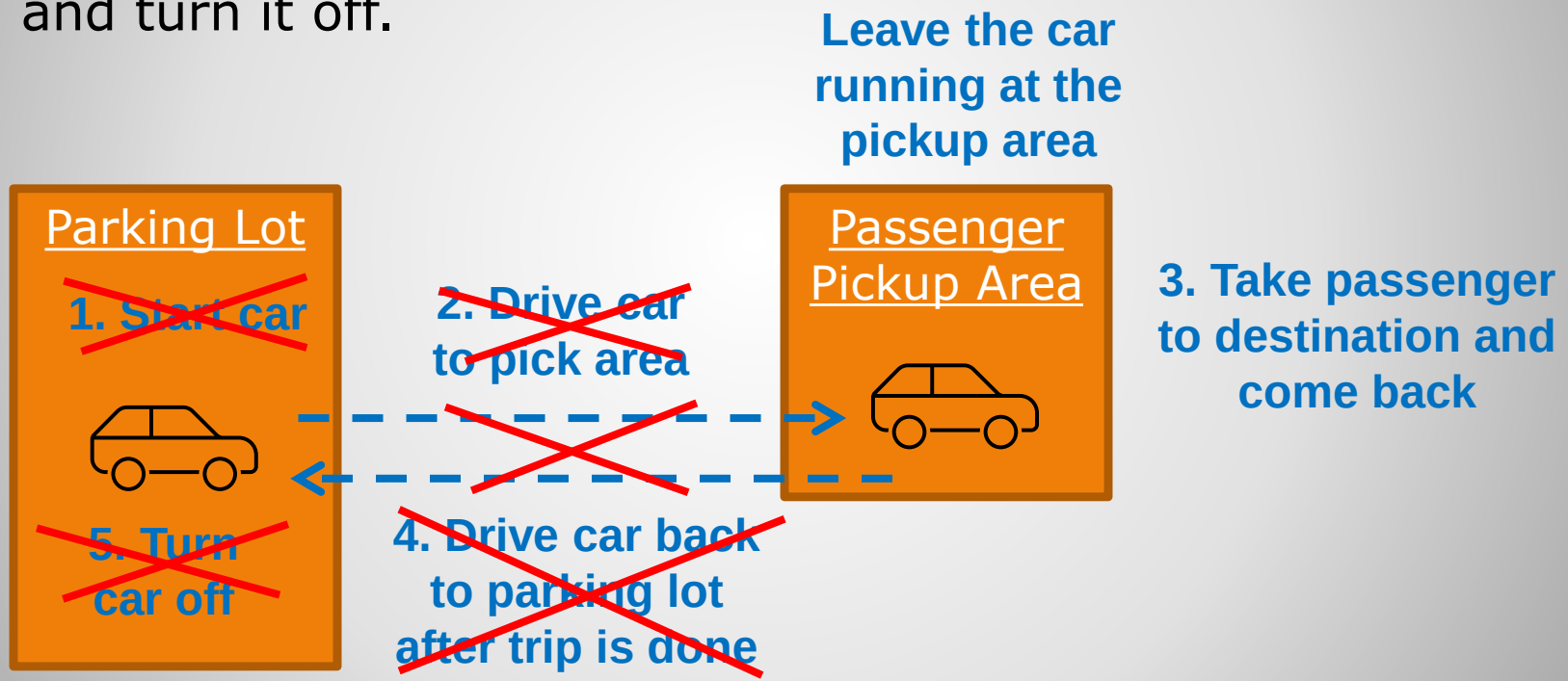
Higher-level Thread Usage

- Every time a passenger wants a ride, a car must be started in the parking lot and driven to the pickup area.
- When the car is done being used it must be put in the parking lot and turned off.
- There is wasted time starting the car, driving it to the pickup area, driving the car back to the parking lot, and turning it off.



Car Service Example

- It would be faster if we could just leave the car running at the passenger pickup area.
- If we did this then we would not have to start the car, drive it to the pickup area, drive the car back to the parking lot, and turn it off.



Faster Car Service Example

- Using threads is similar to the car service example (threads are like the cars).
- Creating and destroying threads is computationally expensive (just like it takes a long time to start the car and drive it to the pickup area, drive it back to the parking lot, and turn it off).
- Instead of always creating and destroying threads we can reuse them and leave them "running".
- A thread pool allows us to reuse threads.
- Using normal threads is like the first car service example.
- A thread pool is like the faster car service example.

Thread Pool

Executor and Thread Pools

- Java's **Executor** is a thread pool implementation.
- Executors allow you to run threads and asynchronous tasks but hide some details and operate more efficiently.
- Thread creation and destruction are computationally expensive.
- A big advantage of using an Executor (a thread pool) is its threads can be easily reused. This will greatly enhance performance.
  - Threads not destroyed - When a thread in the thread pool finishes it is not destroyed (it is kept around and reused by another task in the future)
  - Minimal thread creation - When a thread is needed it does not have to be created (most of the time).

Executors

- Executors were introduced in Java 8.
- Hides thread creation details.
- Link: https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/
- The Executor manages a thread pool for us (hides thread creation details).

Executors and ExecutorService

- **Executors Class** – The Executors class contains methods to create and manage an Executor instance. All methods on the Executors class are static.
- **Executor Interface** – Contains one method named execute that accepts a Runnable. When execute is called the Executor will assign the Runnable to an available thread in the thread pool (or create a new thread if necessary).
- **ExecutorService Interface** – Extends the Executor interface. It contains methods to manage the lifecycle of an Executor.
- The Exectutors class has methods to create instances of ExectutorService. Some example methods are:
  - newSingleThreadExecutor() – Only one thread runs. The thread is reused.
  - newFixedThreadPool() – Fixed number of threads that are always there. Threads are reused.
  - newCachedThreadPool() – Reuses threads but those threads do not stay if they are unused (terminates and removes after a short period of time). No limit on the number of threads that can be created.
  - and so on…
- Here is an example…

Thread Example using Executors and ExecutorService

- The following code creates an instance of an ExecutorService and runs code on a thread within that ExecutorService:

Create the ExectutorService instance

```
ExecutorService exec = Executors.newSingleThreadExecutor();
```

Run code on a thread within the ExectuorService instance (the code in the { } will be run on another thread)

```
exec.submit( () -> {
   String tName = Thread.currentThread().getName();
   System.out.printf("Message from %s\n", tName);
});
// Other code here…

exec.shutdown();
```

Shutdown the ExecutorService when you are done using threads. Only do this when you are completely done with using threads. Creating and destroying it would mean creating and destroying lots of threads which defeats the purpose of using it in the first place.

Executor Starting a Runnable Class

- The example below runs an instance of a class that implements Runnable using the executor.

```
Class MyRunnable implements Runnable {
   @Override
   public void run() {
      String tName = Thread.currentThread().getName();
      System.out.printf("Message from %s\n", tName);
   }
}

ExecutorService exec = Executors.newSingleThreadExecutor();
MyRunnable mr = new MyRunnable();
exec.submit(mr);
// Other code here…

// Eventually you should shutdown the executor.
exec.shutdown();
```

Create a class that implements Runnable

Create a new instance of MyRunnable and run on the executor (submit will call run on mr)

Wait for Completion of Executor Threads

- To wait for completion of all threads you can call the awaitTermination method on the Executor.
- For example (assume this code is in the main thread):

Stops executor from taking new tasks

Wait one minute for running tasks to finish

```
exec.shutdown();
try {
   exec.awaitTermination(1, TimeUnit.MINUTES);
} catch (InterruptedException ex) {
   Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
```

- **shutdown** will not let the executor take on new tasks. The main thread will not block when shutdown is called. If you call awaitTermination (after shutdown) the main thread will wait for the executor to finish for the specified amount of time.
- **shutdownNow** also tries to stop the executor's running threads (as well as not taking on new tasks).

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit( () -> {
   String threadName =
   Thread.currentThread().getName(); System.out.println("Hello " +
   threadName);
});

try {
   System.out.println("attempt to shutdown executor");
   executor.shutdown();
   executor.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException e) {
   System.err.println("tasks interrupted");
} finally {
   if ( !executor.isTerminated() ) {
      System.err.println("cancel non-finished tasks");
   }
   executor.shutdownNow();
   System.out.println("shutdown finished");
}
```

Run code on another thread (uses lambda expression)

Shutdown the executor and wait 5 seconds for all threads to finish

Check if all threads finished. If threads are still running print an error message.

Stop the executor

Executor Shutdown Example

JavaFX GUI and ExecutorService Shutdown

- You can define a clean up method in a controller class and have it called automatically when the current window is closing.

```
class MyController {
   public void cleanup() {
      System.out.println("cleanup called on MyController...");
      // Code to run the ExecutorService shutdown goes here...
   }
}
```

Write a cleanup method in the controller class

Replace the stage creation code in the start method as follows (in App.java):

```
FXMLLoader loader = new FXMLLoader(getClass().getResource("primary.fxml"));
Parent root = loader.load();
MyController myController = loader.getController();
Scene scene = new Scene(root);
stage = new Stage();
stage.setScene(scene);
stage.setOnHidden(e -> myController.cleanup());
stage.show();
```

Add the controller's clean up method as an event handler for when the window is being hidden

Waiting on Multiple Threads to Finish - CountDownLatch

- Wait until multiple threads have finished their work.
- Use a countdown latch to manage waiting on multiple threads.
- A countdown latch maintains a count which you will initialize to some value.
- When the count reaches 0 the countdown latch sends out a signal.
- CountDownLatch is thread safe.
- For example:

Main Thread Code

```
final CountDownLatch finishedSignal = new CountDownLatch(10);
// Create and run 10 threads here …
try {
   finishedSignal.await();
} catch (InterruptedException ex) {
}
```

Main thread waits for the CountDownLatch instance to reach 0. It will not move beyond await until the finishedSignal CountDownLatch instance reaches 0.

Thread code

```
// Do work in thread here…
finishedSignal.countDown();
```

Each thread calls the countDown method to decrement the CountDownLatch's counter when it is finished

Waiting on Multiple Threads to Finish - CountDownLatch

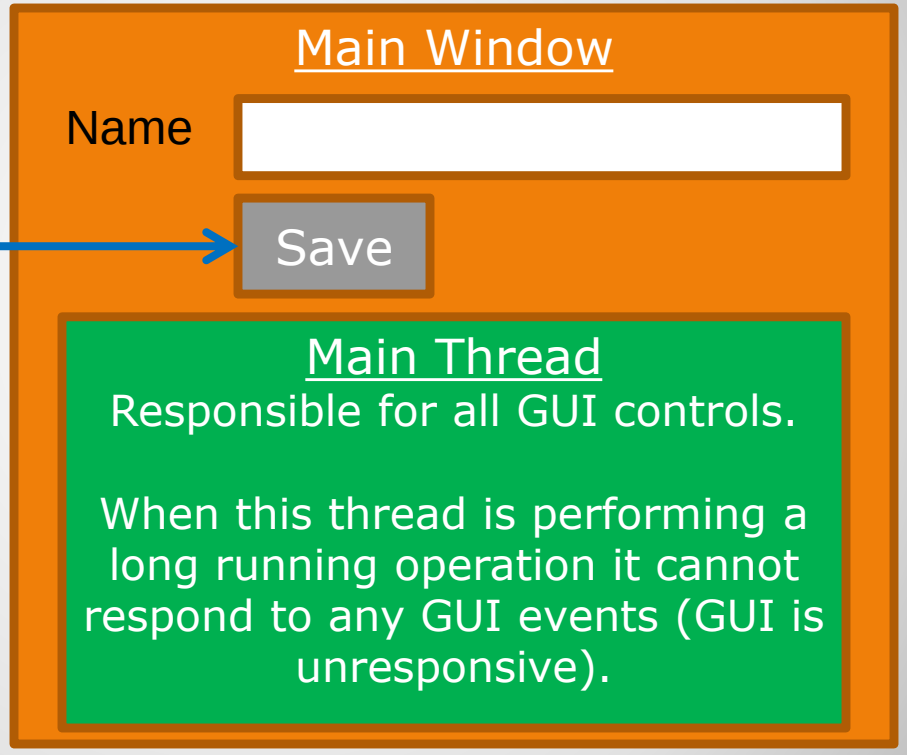
- Now we will discuss using threads in a JavaFX application…

JavaFX and Threads

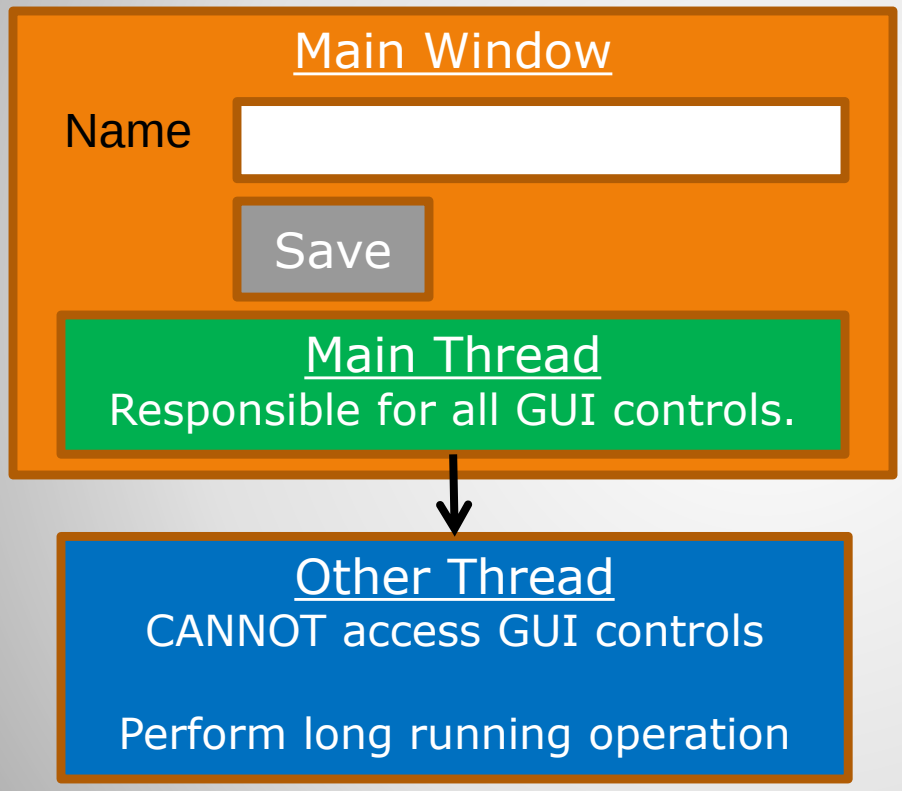
Long Running Operation in GUI

- The main thread in a JavaFX application is responsible for all GUI controls.
- A JavaFX GUI becomes unresponsive if a long running operation takes place.
- This means that if the user clicks any controls in the GUI, it will not respond.

Button will be unresponsive if a long running operation is taking place



- A solution to the "hanging" GUI problem is to use other threads to perform long running operations (GUI will not hang).
- Unfortunately, this creates other problems because only the main GUI thread can update the controls in the window.



The main thread creates another thread to perform the long running operation

The other thread may need to change values in the GUI controls, but it cannot because it does not have access to them

Other Thread runs Long Operation

Update GUI from Other Thread

- The other thread can ask the main thread to update the GUI controls on its behalf.
- Use the **Platform.runLater()** method to update GUI controls from other threads.

```
Platform.runLater( () -> myTextField.setText("abc") );
```

Code to run on the main thread

- The Platform.runLater method takes a Runnable instance and passes it to the main thread.
- The main thread adds the Runnable instance to an event queue (it will run it at some point on the other thread's behalf).
- The call to runLater in the other thread does NOT block.

- End of slides

End of Slides