Object Oriented Programming with JAVA

Module 5-Multithreaded Programming

Dr. Vinoth Kumar M , Associate Professor
Dept. of Information Science & Engineering, RVITM

Dr. Kirankumar K , Assistant Professor Dept. of Information Science & Engineering, RVITM

Tasks and Threads

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread
  - Java libraries use the Runnable interface
  - work done by method run()
- **Thread**: a Java class for a thread
  - work done by method run()

- How to associate a task with a thread?
- How to start a thread?

Creating a Task and Thread

- Warning: old way(s), new ways
- First, if you have a thread object, you can call start() on that object
  - Makes it available to be run
  - When it's time to run it, Thread's run() is called
- So, create a thread using "old" (not good) way
  - Write class that extends Thread, e.g. MyThread
  - Define your own run()
  - Create a MyThread object and call start() on it

Runnables and Thread

- Use the “task abstraction” and create a class that implements Runnable interface
  - Define the run() method to do the work you want
- Now, two ways to make your task run in a separate thread
  - Create a Thread object and pass a Runnable to the constructor
  - As before, call start() on the Thread object

Do we need a Thread "manager"?

- If your code is responsible for creating a bunch of tasks, linking them with Threads, and starting them all, then you have things to worry about:
  - What if you start too many threads? Can you manage the number of running threads?
  - Can you shutdown all the threads?
  - If one fails, can you restart it?

Executors

- An Executor is an object that manages running tasks
  - Submit a Runnable to be run with Executor's execute() method
  - So, instead of creating a Thread for your Runnable and calling start() on that, do this:
    - Get an Executor object, say called exec
    - Create a Runnable, say called myTask
    - Submit for running: exec.execute(myTask)

How to Get an Executor

- Use static methods in Executors library.
- Fixed "thread pool": at most N threads running at one time

```
Executor exec =
    Executors.newFixedThreadPool(MAX_THREADS);
```

- Unlimited number of threads

```
Executor exec =
    Executors.newCachedThreadPool();
```

Summary So Far

- Create a class that implements a Runnable to be your “worker”
- Create Runnable objects
- Create an Executor
- Submit each Runnable to the Executor which starts it up in a separate thread

Synchronization

- Understand the issue with concurrent access to shared data?
  - Data could be a counter (int) or a data structure (e.g. a Map or List or Set)
- A critical section: a block of code that can only be safely executed by one thread at a time
- A lock: an object that is "held" by one thread at a time, then "released"

Synchronization in Java (1)

- Any object can serve as a lock
  - Separate object: `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
        // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others "block" or wait until the first thread releases the lock

Synchronized Methods

- Common situation: all the code in a method is a critical section
  - I.e. only one thread at a time should execute that method
  - E.g. a getter or setter or mutator, or something that changes shared state info (e.g. a Map of important data)
- Java makes it easy: add synchronized keyword to method signature. E.g.

```
public synchronized void update(...)
```

Summary So Far

- Concurrent access to shared data
  - Can lead to serious, hard-to-find problems
  - E.g. race conditions
- The concept of a lock
- Synchronized blocks of code or methods
  - One thread at a time
  - While first thread is executing it, others block

More Advanced Synchronization

- A semaphore object
  - Allows simultaneous access by N threads
  - If N==1, then this is known as a mutex (mutual exclusion)
  - Java has a class Semaphore
- Java class CountDownLatch
  - Created with a count (often a number of "worker" threads). Say object is `allWorkersDone`
  - Another thread (a "manager") waits for all the workers to call countDown() on that object

Barriers

- Java class CyclicBarrier
  - A rendezvous point or barrier point
  - Worker threads wait at a spot until all get there
  - Then all proceed

Using CountDownLatch

- Here are some common scenarios and demo programs for them

- You'll use the last of these for the War card-game program!

Scenario #1

- A “manager” thread and N “worker” threads
- Manager starts workers but then must wait for them to finish before doing follow-up work
- Solution:
  - Manager creates a CountDownLatch with value N
  - After workers starts, manager calls await() on that
  - When each worker completes its work, it calls countDown() on the latch
  - After all N call countDown(), manager is un-blocked and does follow-up work
- Example use: parallel divide and conquer like mergesort
- Code example: SyncDemo0.java

Scenario #2

- A “manager” thread and N “worker” threads
- Manager starts workers but wants them to “hold” before doing real work until it says “go”
- Solution:
  - Manager creates a CountDownLatch with value 1
  - After each workers start, it calls await() on that Latch
  - At some point, when ready, the manager calls countDown() on that Latch
  - Now Workers free to continue with their work
- Code example: SyncDemo1.java

Scenario #3

- Work done in “rounds” where:
  - All workers wait for manager to say “go”
  - Each worker does its job and then waits for next round
  - Manager waits for all workers to complete a round, then does some follow-up work
  - When that’s done, manager starts next round by telling workers “go”
- Solution: combine the two previous solutions
  - First Latch: hold workers until manager is ready
  - Second Latch: manager waits until workers finish a round
  - Worker’s run() has loop to repeat
  - Manager must manage Latches, recreating them at end of round
- Example use: a card game or anything that has that kind of structure
- Code example: SyncDemo2.java

Enumerated types

- **enum**: A type of objects with a fixed set of constant values.

```
public enum Name{
    VALUE, VALUE, ..., VALUE
}
```

  - Usually placed into its own .java file.
  - C has `enum`s that are really `int`s; Java's are objects.

```
public enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}
```

What is an enum?

- The preceding `enum` is roughly equal to the following short class:

```
public final class Suit extends Enum<Suit> {
    public static final Suit CLUBS    = new Suit();
    public static final Suit DIAMONDS = new Suit();
    public static final Suit HEARTS   = new Suit();
    public static final Suit SPADES   = new Suit();

    private Suit() {}   // no more can be made
}
```

What can you do with an enum?

- use it as the type of a variable, field, parameter, or return

```
public class Card {
    private Suit suit;
    ...
}
```

- compare them with `==` (why don't we need to use `equals`?)

```
if (suit == Suit.CLUBS) { ...
```

- compare them with `compareTo` (by order of declaration)

```
}
```

The switch statement

```
switch (boolean test) {
    case value:
        code;
        break;
    case value:
        code;
        break;
    …
    default:   // if it isn't one of the above values
        code;
        break;
}
```

- an alternative to the `if/else` statement
  - must be used on integral types (e.g. `int`, `char`, `long`, **`enum`**)

Enum methods

method	description
`int compareTo(E)`	all `enum` types are Comparable by order of declaration
`boolean equals(o)`	not needed; can just use `==`
`String name()`	equivalent to `toString`
`int ordinal()`	returns an enum's 0-based number by order of declaration (first is 0, then 1, then 2, ...)

method	description
`static E valueOf(s)`	converts a string into an `enum` value
`static E[] values()`	an array of all values of your enumeration

More complex enums

- An enumerated type can have fields, methods, and constructors:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private int cents;

    private Coin(int cents) {
        this.cents = cents;
    }
    public int getCents()  { return cents; }

    public int perDollar() { return 100 / cents; }

    public String toString() {    // "NICKEL (5c)"
        return super.toString() + " (" + cents + "c)";
    }
}
```