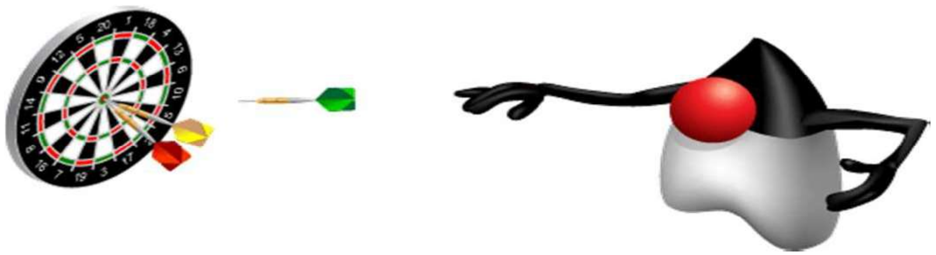
16

Multi Threading

Objectives

After completing this lesson, you should be able to:

- Describe operating system task scheduling
- Create worker threads using `Runnable` and `Callable`
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and concurrent atomic to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections



Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- **Processes:** A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- **Thread:** A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Legacy `Thread` and `Runnable`

Prior to Java 5, the `Thread` class was used to create and start threads. Code to be executed by a thread is placed in a class, which does either of the following:

- Extends the `Thread` class
  - Simpler code
- Implements the `Runnable` interface
  - More flexible
  - `extends` is still free.

Extending `Thread`

Extend `java.lang.Thread` and override the `run` method:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Starting a `Thread`

After creating a new `Thread`, it must be started by calling the `Thread`'s `start` method:

```
public static void main(String[] args) {
    ExampleThread t1 = new ExampleThread();
    t1.start();
}
```

Schedules the run method to be called

Implementing `Runnable`

Implement `java.lang.Runnable` and implement the `run` method:

```
public class ExampleRunnable implements Runnable {
    private final String name;
    public ExampleRunnable(String name) {
        this.name = name;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(name + ":" + i);
        }
    }
}
```

Executing `Runnable` Instances

After creating a new `Runnable`, it must be passed to a `Thread` constructor. The `Thread`'s `start` method begins execution:

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
}
```

A `Runnable` with Shared Data

Static and instance fields are potentially shared by threads.

```
public class ExampleRunnable implements Runnable {
    private int i;

    @Override
    public void run() {
        for(i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Potentially shared variable

One Runnable: Multiple Threads

An object that is referenced by multiple threads can lead to instance fields being concurrently accessed.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    Thread t2 = new Thread(r1);
    t2.start();
}
```

A single `Runnable` instance

Detecting Interruption

Interrupting a thread is another possible way to request that a thread stop executing.

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread started");
        while(!Thread.interrupted()) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

static `Thread` method

Interrupting a Thread

Every thread has an `interrupt()` and `isInterrupted()` method.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    t1.interrupt();
}
```

Interrupt a thread

Thread.sleep()

A `Thread` may pause execution for a duration of time.

```
long start = System.currentTimeMillis();
try {
    Thread.sleep(4000);
} catch (InterruptedException ex
    // What to do?
}
long time = System.currentTimeMillis() - start;
System.out.println("Slept for " + time + " ms");
```

`interrupt()` called while sleeping

Additional `Thread` Methods

- There are many more `Thread` and threading-related methods:
  - `setName(String)`, `getName()`, and `getId()`
  - `isAlive()`: Has a thread finished?
  - `isDaemon()` and `setDaemon(boolean)`: The JVM can quit while daemon threads are running.
  - `join()`: A current thread waits for another thread to finish.
  - `Thread.currentThread()`: `Runnable` instances can retrieve the `Thread` instance currently executing.
- The `Object` class also has methods related to threading:
  - `wait()`, `notify()`, and `notifyAll()`: Threads may go to sleep for an undetermined amount of time, waking only when the `Object` they waited on receives a wakeup notification.

Methods to Avoid

Some `Thread` methods should be avoided:

- `setPriority(int)` and `getPriority()`
  - Might not have any impact or may cause problems
- The following methods are deprecated and should never be used:
  - `destroy()`
  - `resume()`
  - `suspend()`
  - `stop()`

Deadlock

Deadlock results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
   synchronized(obj2) {
   }
}
```

Thread 1 pauses after locking obj1's monitor.

```
synchronized(obj2) {
   synchronized(obj1) {
   }
}
```

Thread 2 pauses after locking obj2's monitor.

The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

Recommended Threading Classes

Traditional `Thread` related APIs are difficult to code properly. Recommended concurrency classes include:

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7

java.util.concurrent.ExecutorSe

An `ExecutorService` is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
- Tasks can be:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- Implementing instances can be obtained with `Executors`.

```
ExecutorService es = Executors.newCachedThreadPool();
```

Example ExecutorService

This example illustrates using an `ExecutorService` to execute `Runnable` tasks:

```
package com.example;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        es.execute(new ExampleRunnable("one"));
        es.execute(new ExampleRunnable("two"));
        es.shutdown();
    }
}
```

Execute this Runnable task sometime in the future

Shut down the executor

Shutting Down an `ExecutorService`

Shutting down an `ExecutorService` is important because its threads are nondaemon threads and will keep your JVM from shutting down.

Stop accepting new `Callables`.

If you want to wait for the `Callables` to finish

```
es.shutdown();
try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

java.util.concurrent.Callable

The `Callable` interface:

- Defines a task submitted to an `ExecutorService`
- Is similar in nature to `Runnable`, but can:
  - Return a result using generics
  - Throw a checked exception

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

Example `Callable` Task

```
public class ExampleCallable implements Callable {

  private final String name;
  private final int len;
  private int sum = 0;

  public ExampleCallable(String name, int len) {
    this.name = name;
    this.len = len;
  }

  @Override
  public String call() throws Exception {
    for (int i = 0; i < len; i++) {
      System.out.println(name + ":" + i);
      sum += i;
    }
    return "sum: " + sum;
  }
}
```

Return a String from this task: the sum of the series

java.util.concurrent.Future

The `Future` interface is used to obtain the results from a `Callable`'s `V call()` method.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
    V result = future.get();
} catch (ExecutionException|InterruptedException ex) {
}
```

`ExecutorService` controls when the work is done.

Gets the result of the `Callable`'s `call` method (blocks if needed).

If the `Callable` threw an `Exception`

Example

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

  ExecutorService es = Executors.newFixedThreadPool(4);
  Future<String> f1 = es.submit(new ExampleCallable("one",10));
  Future<String> f2 = es.submit(new ExampleCallable("two",20));

  try {
    es.shutdown();
    es.awaitTermination(5, TimeUnit.SECONDS);
    String result1 = f1.get();
    System.out.println("Result of one: " + result1);
    String result2 = f2.get();
    System.out.println("Result of two: " + result2);
  } catch (ExecutionException | InterruptedException ex) {
    System.out.println("Exception: " + ex);
  }

}
```

Wait 5 seconds for the tasks to complete

Get the results of tasks f1 and f2

Threading Concerns

- Thread Safety
  - Classes should continue to behave correctly when accessed from multiple threads.
- Performance: Deadlock and livelock
  - Threads typically interact with other threads. As more threads are introduced into an application, the possibility exists that threads will reach a point where they cannot continue.

Shared Data

Static and instance fields are potentially shared by threads.

```
public class SharedValue {
    private int i;

    // Return a unique value
    public int getNext() {
        return i++;
    }
}
```

Potentially shared variable

Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - “Safely” accessing shared fields is your responsibility.

Two threads accessing an instance of the `SharedValue` class might produce the following:

```
i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...
```

Zero produced twice

Out of sequence

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

- Local variables
- Method parameters
- Exception handler parameters
- Immutable data

The `synchronized` Keyword

The `synchronized` keyword is used to create thread-safe code blocks. A `synchronized` code block:

- Causes a thread to write all of its changes to main memory when the end of the block is reached
- Is used to group blocks of code for exclusive execution
  - Threads block until they can get exclusive access
  - Solves the atomic problem

synchronized Methods

```java
public class SynchronizedCounter {
  private static int i = 0;

  public synchronized void increment(){
    i++;
  }

  public synchronized void decrement(){
    i--;
  }
  public synchronized int getValue(){
    return i;
  }
}
```

synchronized Blocks

```java
public void run(){
  for (int i = 0; i < countSize; i++){
    synchronized(this){
      count.increment();
      System.out.println(threadName
             + " Current Count: " + count.getValue());
    }
  }
}
```

Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.

```
synchronized ( this ) { }
```

- `synchronized` blocks can be nested.

Threading Performance

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

Performance Issue: Examples

- **Deadlock** results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
   synchronized(obj2) {
   }
}
```

Thread 1 pauses after locking obj1's monitor.

```
synchronized(obj2) {
   synchronized(obj1) {
   }
}
```

Thread 2 pauses after locking obj2's monitor.

- **Starvation** and **Livelock**

Summary

In this lesson, you should have learned how to:

- Describe operating system task scheduling
- Use an `ExecutorService` to concurrently execute tasks
- Identify potential threading problems
- Use `synchronized` and concurrent atomic to manage atomicity
- Use monitor locks to control the order of thread execution
- Use the `java.util.concurrent` collections