Concurrency
(with multithreading)

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team

https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaThreadsSamples
http://101companies.org/wiki/Contribution:javaMultithreading
http://101companies.org/wiki/Contribution:javaMessaging
Introduction
Context: **Parallel computing**

Wikipedia as of 21 June 2011: “**Parallel computing** is a form of **computation** in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved **concurrently** ("in parallel"). There are several different forms of parallel computing: **bit-level**, **instruction level**, **data**, and **task parallelism**.”
Why parallelism?

- Many users of one server
- Many tasks on one machine
- Large data volume
- Responsive UIs
- ...

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Related concepts

- Parallel computing
- Distributed computing
- Concurrent computing
Parallel programming:
- There are several (possibly virtual) processors.
- Forms of parallelism
  - Task parallelism -- distribute execution processes
  - Data parallelism -- distribute data across parallel computing nodes

Distributed programming: several systems, ...
Concurrent programming: parallelism + interaction
Multithreading: Java’s approach to task parallelism
Multitasking: OS-level parallelism
Basics of multithreading
Running example: compute the sum of the lengths of an array of strings.

```java
private static int sumLength(String[] a) {
    int sum = 0;
    for (String s : a) {
        sum += s.length();
    }
    return sum;
}
```
Thread-based parallelism with callable and futures (in fact, functors)

DEMO

See package sumlength.sequential (reference implementation).
See package sumlength.parallel (multithreaded implementation).
Key concepts

- Threads -- threads of execution
- Thread pools -- resource management
- **Runnables** -- functor for thread’s functionality
- **Callables** -- functors for threads with results
- **Futures** -- objects for result access from threads
- **Objects** -- wait/notify for threads on objects.

Everything is an object.
Java threads

- ... are objects.
- ... model *threads of execution*.
- ... can be ...
  - ... started,
  - ... interrupted,
  - ... sent to *sleep*,
  - ... made *wait for notification*, etc.
- ... can be constructed by ...
  - ... passing a “*Runnable*” to Thread’s constructor,
  - ... instantiating a subclass of *Thread*.
Life Cycle of a Thread

We will not discuss some lock/sleep-related details.

http://www.tutorialspoint.com/java/java_multithreading.htm
Thread pools

- We can construct individual thread objects.
- We may also use thread pools. Here is why:
  - Thread objects are expensive.
  - Threads in pools can be recycled.
  - Pools minimize the overhead of thread creation.
  - Pools also lead to systems that degrade gracefully.
Runnables

- Serves to capture a thread’s functionality
  
  ```java
  new Thread(new MyRunnable())
  ```

- Functor interface
  
  ```java
  void run()
  ```
Callable

Serves to capture a thread’s functionality where the thread is supposed to return a result.

Functor interface

V call() throws Exception;
Futures

- Asynchronous composition:
  - The producer (thread) returns a value eventually.
  - The consumer needs to receive a value eventually.
  - To this end, thread submission returns a Future.
    - ... an object from which to read the result:

      \[ V \text{ get()} \]
Concurrency
Terminology cont’d

Wikipedia as of 21 June 2011: “Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes.”
Dining philosophers (standard example of concurrent programming)


http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Dining philosophers (standard example of concurrent programming)

- Due to: Edsger Dijkstra and Tony Hoare, 1971
- 5 philosophers are sitting at a round table with a bowl of spaghetti in the centre. They either eat or think. While eating, they are not thinking, and while thinking, they are not eating.
- To eat, each one needs 2 forks. There are 5 forks total, one between any adjacent two philosophers.

How to model this problem say in Java?
Multithreaded dining philosophers

- Philosophers are modeled as threads.
- Forks are essentially shared resources.
An unsuccessful attempt at the dining philosophers

DEMO

See package dining.naive.
See package dining.threadpool (a variation with thread pools).
Problems

- 2 philosophers may grab the same fork simultaneously.
  - This should not be allowed.
    - We will leverage Java’s “synchronization” to this end.
- All philosophers may hold on just 1 fork forever.
  - This is called a deadlock.
    - No progress; threads are waiting for one another.
      - This requires a problem-specific strategy.
Java’s synchronization

```java
public class SynchronizedCounter {
    private int c = 0;

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

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
Synchronization

DEMO

- Package banking
  - Account with concurrent access
In need of object monitors

```java
public synchronized void take() {
    while(taken) { }
    taken = true;
}
```

```java
public synchronized void drop() {
    taken = false;
}
```

What’s the problem here?
Object monitors at work

```java
class Monitor {
    private boolean taken;

    public synchronized void take() throws InterruptedException {
        while (taken)
            wait();
        taken = true;
    }

drop() releases monitor if taken.

    public synchronized void drop() {
        taken = false;
        notifyAll();
    }

drop() wakes up all waiting threads.
}
```

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Object monitors

DEMO

- Package dining.simple
  - Synchronization for dining philosophers
Characteristics of deadlocks

- Shared reusable mutually exclusive resources
- Incremental acquisition of resources
- No pre-emption (confiscation)
- Idle waiting — no useful actions
Options for improved dining philosophers

- Time out
  - Return 1st fork if 2nd cannot be claimed.
  - This may lead to *starvation*: there is no guarantee that every philosopher will be getting food eventually (i.e., a thread is perpetually denied resources).

- Stabilize
  - Have one left-handed philosopher. Our actual implementation still has problems.
Further attempts at the dining philosophers

DEMO

See package dining.timeout.
See package dining.stabilized.
Resource management

- Limited resources
  - e.g.: Network connections
- Unbounded requests
  - e.g.: http requests
- Ensure limit
  - Semaphore (monitor)
Semaphore

- A semaphore maintains a set of permits.
- Each `acquire()` blocks if necessary until a permit is available, and then takes it.
- Each `release()` adds a permit, potentially releasing a blocking acquirer.
- However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.
- Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.
Semaphores

DEMO

See package dining.semaphore.
Concurrency / parallelism for 101system

DEMO

http://101companies.org/wiki/Contribution:javaMultithreading
Messaging-based concurrency / distribution
Concurrency / parallelism for 101system

DEMO

http://101companies.org/wiki/Contribution:javaMessaging
Summary

- Important Java concepts
  - Thread, ThreadPool, Runnable, Callable, Future, synchronized, wait, notifyAll, ...

- Important programming concepts
  - Threads, synchronization, semaphore, ...

- There are much more options (concepts, technologies) out there for concurrency. This was just a tip on the iceberg.