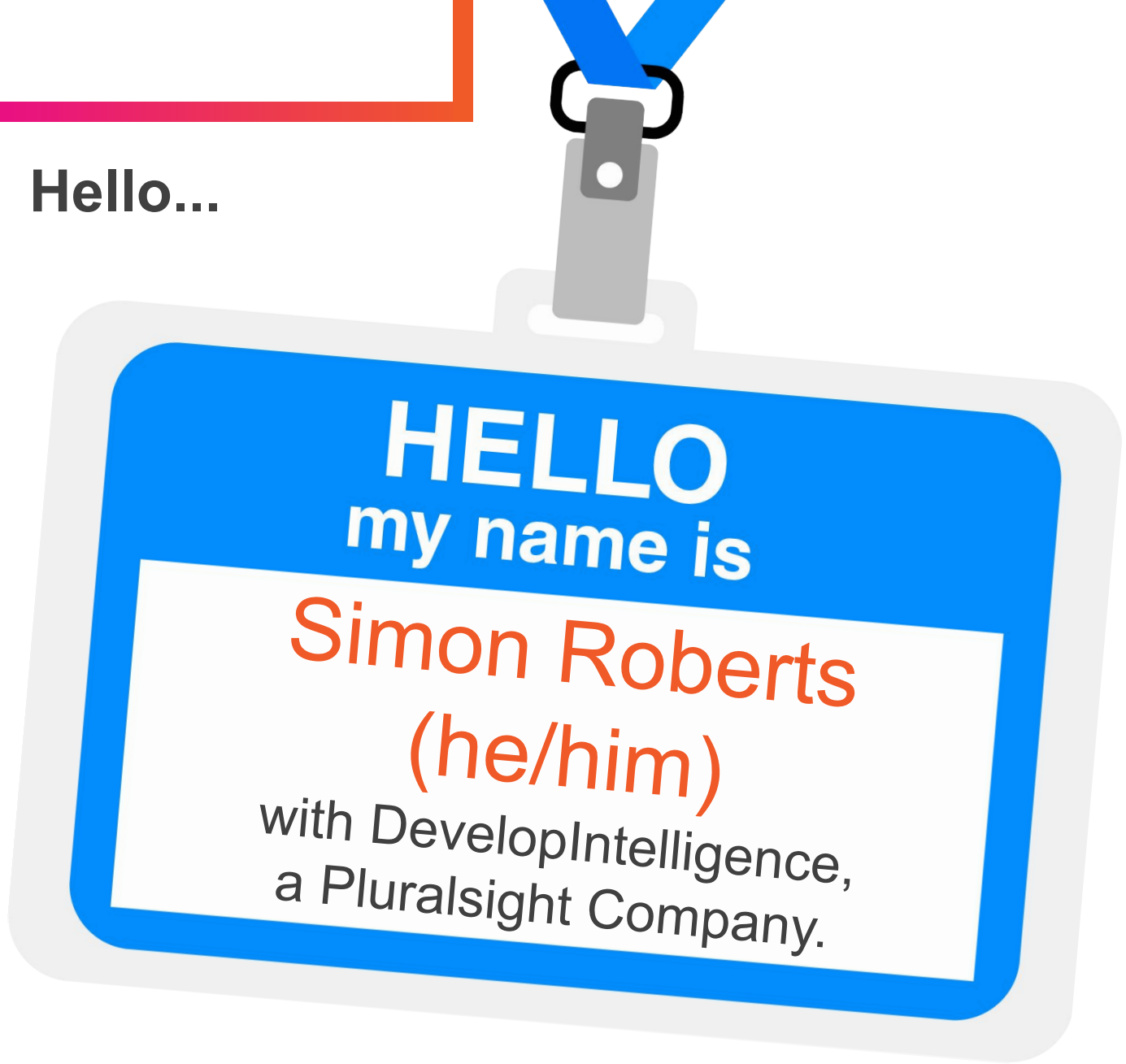
Welcome

Advanced Java TT
Java Concurrency Primitives

DevelopIntelligence
A PLURALSIGHT COMPANY

Hello...



About me...

Prerequisites

This course assumes you

- Good understanding of the Java programming language to Java 8

Why study this subject?

- Modern CPUs are gaining throughput by getting more cores
- Threads are necessary for a program to benefit from those cores
- Threading has nasty traps for the unwary and knowing how to avoid them is important

We teach over 400 technology topics

You experience our impact on a daily basis!

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Create an inclusive learning environment
- Use an on-screen timer for breaks

…also, if you have an accessibility need, please let me know

Objectives

At the end of this course you will be able to:

- At the end of this course you will be able to:
- Describe the happens-before relationship and its importance
- Use Java's low-level thread control primitives
  - volatile
  - synchronized
  - wait/notify/notifyAll
  - interrupts

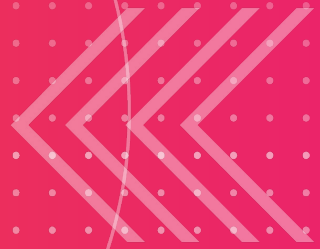
How we’re going to work together

- Discussions, whiteboard diagrams
- Code examples
- You'll have a copy of all the course materials in github
    - Please note, the git repository will be deleted–clone it if you want it!

Student Introductions

- Job title?
- Where are you based?
- Experience with Java?
- Experience with threads in any language?
- Fun fact?

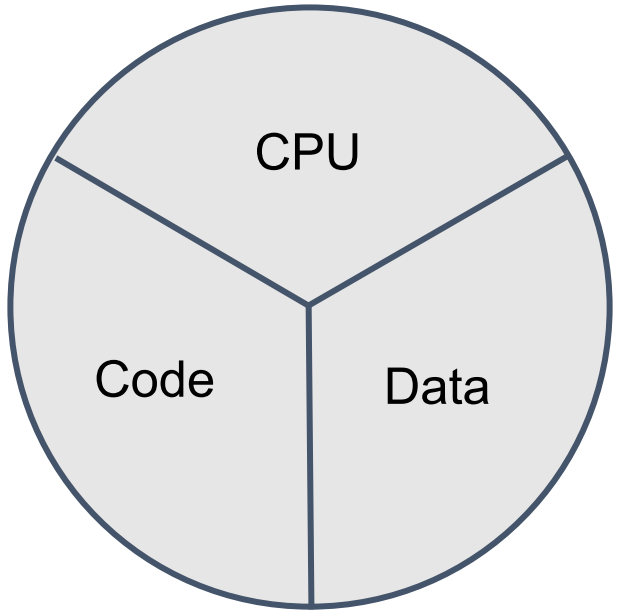
Thank you!



DevelopIntelligence
A PLURALSIGHT COMPANY

Elements of a thread

Silicon, or a thread as a "virtual CPU"



Methods/static methods in objects and classes

Entry point of program:

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

Entry point of thread:

```
public void run()
```

(in an instance of `Runnable`)

Method locals, on the "stack"
or
Objects in the heap

Three problems

1) Visibility
   - Cache and compiler optimizations mean you must ***not*** assume that after one thread writes to a variable, other threads will see the changed data.
   - Do ***not*** assume you understand how the implementation works
2) Transactional integrity
   - read-modify-write cycles
   - partial update of structured data
3) Timing
   - Avoid reading data before it has been published
   - Avoid overwriting data until after other threads have read it
   - Avoid "busy waiting" or "spin waiting" (in the general case)

Visibility is defined by a happens-before relationship

Two actions can be ordered by a ***happens-before*** relationship. If one action *happens-before* another, then the first is visible to the second.

A happens-before relationship ***does not mean*** "*executes prior*", and if the second action does not observe the effect of the first, any assumptions are likely wrong.

Given two actions x and y, hb(x, y) indicates that x happens-before y.

1. If x and y are actions of the ***same thread*** and x comes before y in ***program order***, then hb(x, y).
2. If hb(x, y) and hb(y, z), then hb(x, z).

More happens-before relationships

Develop Intelligence

3. If x writes a volatile variable and y is a subsequent read of the same variable, then hb(x,y).
4. If x unlocks a monitor and y is a subsequent lock of the same monitor, then hb(x,y).
5. If x starts a thread and y is the first action of the thread, then hb(x,y)
6. If x is the last action of a thread and y is another thread observing that the first thread is dead, then hb(x,y).
7. If x interrupts another thread, and y is that thread observing the interrupt, then hb(x,y).
8. If x is the last action of a constructor and y is the first action of the finalizer thread acting on the same object, then hb(x,y).

Monitor locks and synchronized

Develop Intelligence

From a behavioral-model perspective every object has an associated "key", that key can be used to pass through a "gate" that's associated with that same object.

The key is called a monitor lock, and the gate is the ***`synchronized`*** keyword

The `synchronized` keyword may be applied to a block in which case it must be followed by an expression of object type. The "gate" requires the key of that object.

The `synchronized` keyword may be applied to a method. If applied to an instance method, the key must be the one associated with the `this` object. If applied to a static method, the key must be the one associated with the relevant `java.lang.Class` instance.

Passing the `synchronized` "gate"

If the thread already has the necessary key, the thread increments its counter on that key.

If the necessary key is available, the thread takes it, and sets a counter on that key to 1. Taking the key makes the key unavailable to any other thread.

If the necessary key is unavailable, the thread blocks, waiting for the key to be acquired. The thread (for modeling purposes) uses no CPU time while blocked.

When in possession of the key, the thread enters the `synchronized` block.

On exit from the block (by any means) the thread decrements its counter on the relevant key. If the count reaches zero, the key is returned to the owning object.

Using `synchronized` for mutual exclusion

Since at most one thread can be executing code while holding a particular key at any one time, this can be used to create mutual exclusion.

If *all changes* to transactionally sensitive data, *and all reads* of those data are performed subject to the thread holding the key associated with those data, transactional concerns may be addressed.

The effect is entirely dependent on complete and correct use of `synchronized` in all relevant situations

The release of the lock by one thread and subsequent acquisition of that same lock by another also ensures visibility.

Addressing timing

Develop Intelligence

When sharing data, timing issues must also be addressed. Do not try to read data until the producer has prepared it, and do not overwrite it until the consumer has used it.

The two threads will usually be involved in updates that require transactional integrity, and therefore often executing code inside synchronized regions.

"Busy waiting" is wasteful of CPU and should generally be avoided.

Unless the waiting thread releases its monitor lock the other thread will not be able to change the situation that the first is waiting for.

`wait()`, `notify()`, and `notifyAll()` can address these situations.

Effect of `wait()`

Develop Intelligence

The object method wait():

1) Tests for an interrupt, throwing an InterruptedException if found
2) Releases the monitor lock
3) Enters a blocked state, waiting for notification

After it receives notification the thread:

1) Moves to a blocked state, waiting for monitor lock
2) When the lock is acquired, the thread becomes runnable, then when the operating system see fit, running

Note: data should be transactionally stable before calling `wait()`.

Effect of `notify()`

Develop Intelligence

When a thread calls `notify()`, one *randomly chosen* thread that is waiting on the same object receives a notification.

There are almost always two distinct reasons for waiting on the same data structure, but only a single wait. This means that in situations with multiple threads, it's possible to get threads blocked for *both reasons*.

In this situation, a notification can go to a thread that isn't currently able to use it, and *starvation* can result.

Using `notifyAll()` sends a notification to every thread. This avoids the first problem, but creates a *scalability problem*, since every thread must wake, check if it can proceed, and all but one will go back to waiting.

Thread interrupts

Develop Intelligence

Java's thread system includes a "flag-like" notification known as an *interrupt*. This has no relationship to hardware interrupts, but is generally used to ask a thread to shut itself down cleanly.

Threads should never be forcibly shut down from outside, as it's impossible to know what critical operations might be incomplete.

Interrupts can be polled if desired (`aThread.isInterrupted()` and `Thread.interrupted()` methods) , and blocking operations throw InterruptedException. It's the programmer's responsibility to observe the interrupt and shut the thread down cleanly (and reasonably promptly).

Producer-consumer concurrency model

Develop Intelligence

The primitives are hard to use reliably, and it's hard to reason about all the potential consequences of decisions in the general case.

If threads are to cooperate, they must share data in some coordinated way.

A BlockingQueue implementing the "producer-consumer" model provides an architecture that solves all three concurrency problems in a way that is easy to understand implement correctly. Note that the approach has several good architectural qualities, but is often not absolute most efficient.

Producer-consumer essentials

The model can be likened to a production line. At any given instant the work product is either wholly owned by one worker thread, or is inaccessible to all threads and is "in transit" down the production line.

One thread, called the producer prepares data that is wholly owned by that thread (ensure that no thread-shared data are used in this). Because the data are confined to one thread, no concurrency problem can exist.

At the moment the producer has completed its work, it puts the data into a queue and nulls-out its reference to it. At this point, the data is inaccessible to all threads.

Later, another thread (the consumer) takes the data item from the queue. The data are again confined to a single thread, so no concurrency problems can exist.

Behavior of a `BlockingQueue` producer-consumer

When a producer-consumer architecture is implemented correctly using a BlockingQueue the system exhibits certain important properties:

- An object inserted into the queue will be taken by exactly one thread (unless it remains in the queue because nothing tries to take it). This is true even in the face of multiple concurrent producers and consumers.
- If inserting the object into the queue is action x, and taking it from the queue is action y, then hb(x,y). So, *the consumer is guaranteed proper visibility of the data in the object*.

Behavior of a `BlockingQueue` producer-consumer

- If a thread attempts to put an item into the queue when it is full, it will be descheduled and wait for space to be available.
- If a thread attempts to take from an empty queue it will be descheduled and wait for available data.
- Read data items become the exclusive property of the consumer and are not overwritten.
- The combined effect is that *all timing concerns are handled by this architecture*.

Behavior of a `BlockingQueue` producer-consumer

- Data are neither duplicated nor lost (so long as consumers continue to run)
- Items from a single producer will be removed from the queue in the order they were added (though this might be hidden if multiple consumers are running)
- Data are only ever visible to a single thread at any one time, all updates must be completed before the data is transferred through the queue to another thread.
- Because changes are completed entirely in one thread, *all transactional concerns are handled by the architecture*.