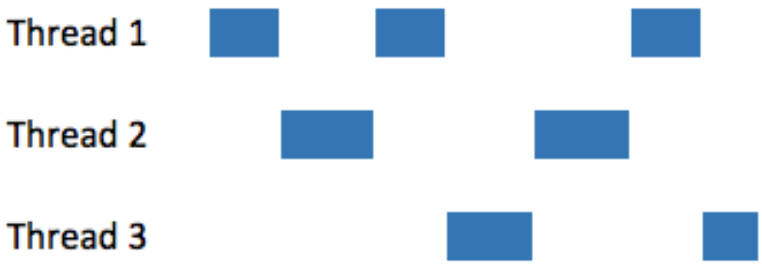
Java Threads

Raul Barbosa

Sistemas Distribuídos 2023/24

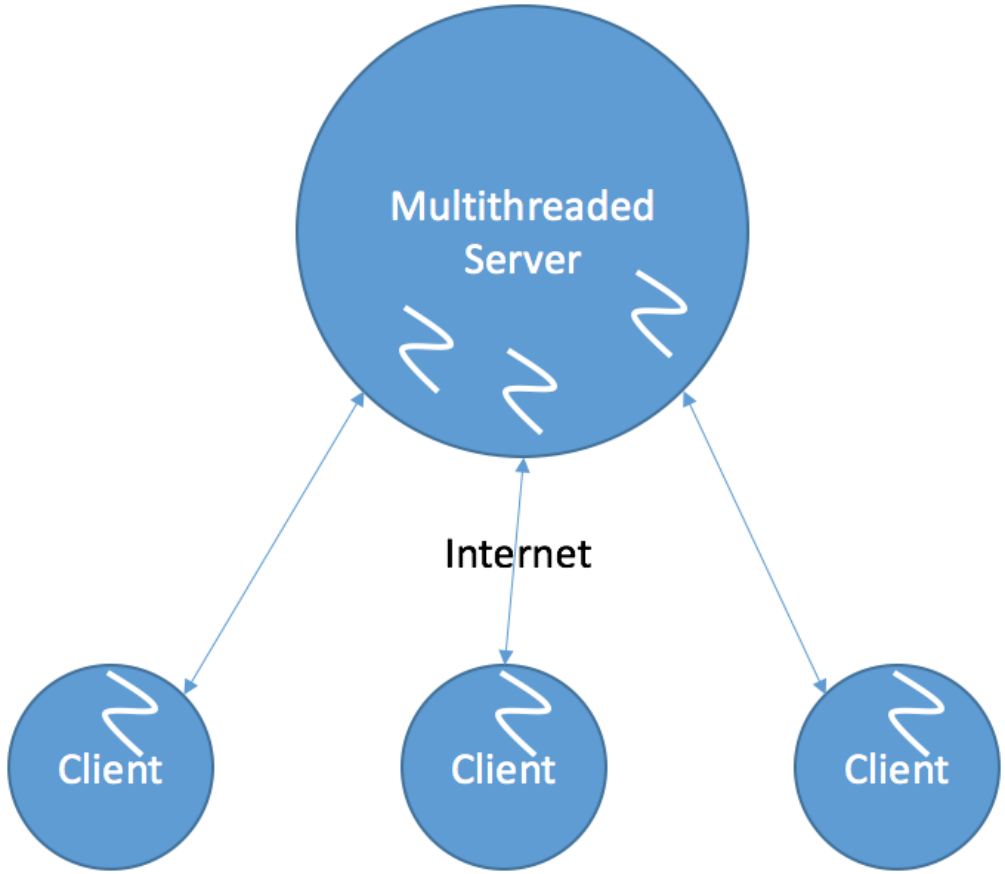
Multithreaded programs – concurrency or parallelism



Time sharing a single processor

Parallel execution on multiple processors

Serving multiple clients using threads



Multithreaded programming in Java

- Threads are built into the Java programming language
- Useful thread synchronization primitives
- Inter-thread communication of events

Two equivalent ways of creating threads

Implement the `Runnable` interface.
Extend the `Thread` class.

Implementing the `Runnable` interface

```
public class MyThread implements Runnable {
    @Override
    public void run() {
        System.out.println("thread started executing");
        // do something useful concurrently...
    }
}

public class Example {
    public static void main(String[] args) {
        MyThread task = new MyThread();   // instantiate MyThread
        Thread thread = new Thread(task); // create a new thread
        thread.start();                   // start the thread -- this will call run()

        // do something else concurrently...
    }
}
```

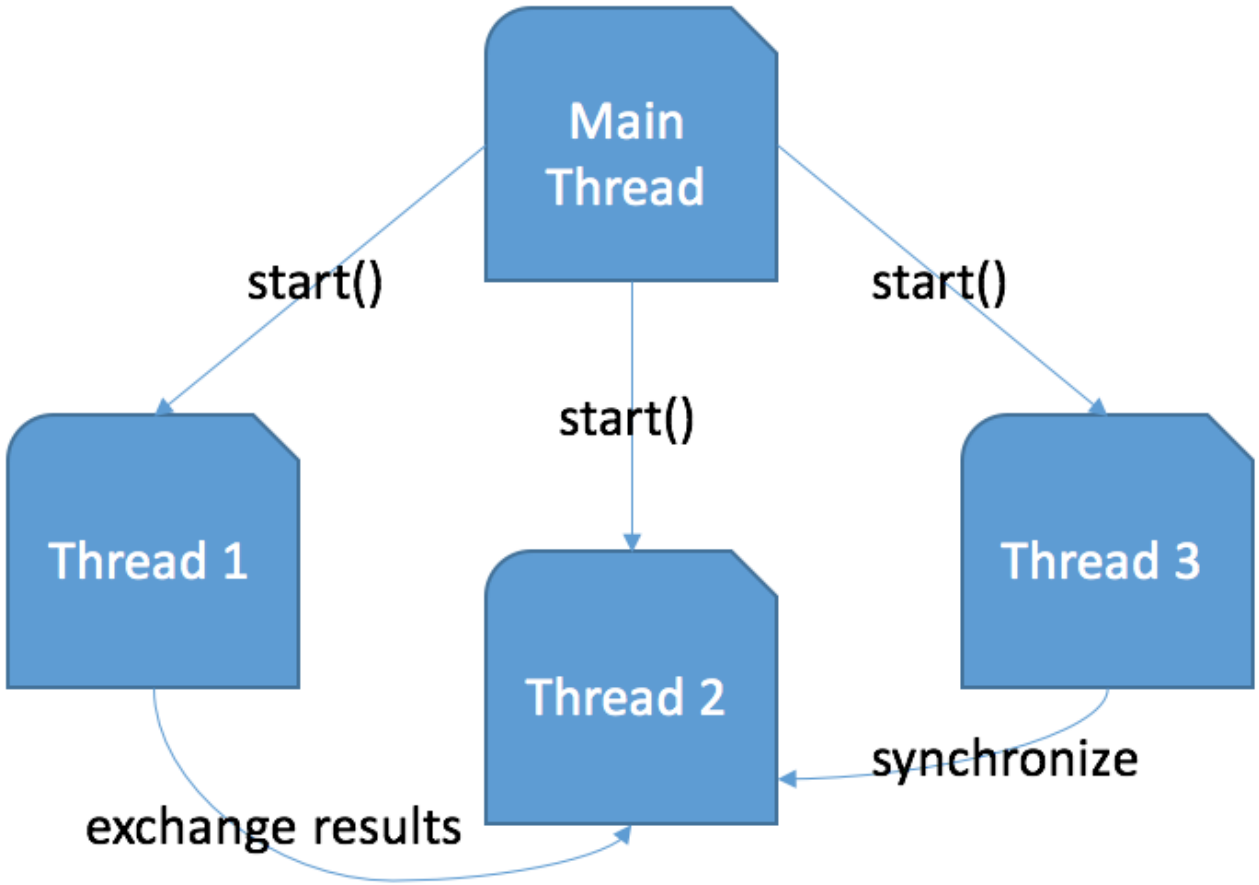
Extending class Thread

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("thread started executing");
        // do something useful concurrently...
    }
}

public class MainClass {
    public static void main(String[] args) {
        MyThread thread = new MyThread(); // instantiate MyThread
        thread.start();                   // start the thread -- this will call run()

        // do something else concurrently...
    }
}
```

The “main” thread may launch numerous threads...



Important methods in class Thread

- `start()` begins a new thread and calls the run() method
- `run()` this method is called by the JVM
- `join()` waits for *this* thread to terminate
- `interrupt()` unblock the thread, throw InterruptedException
- `wait()` wait until another thread calls notify()
- `notify()` wake up one thread waiting on the monitor
- `notifyAll()` wake up all threads waiting on the monitor
- returning from `run()` kills the thread – don't use stop()

Javadocs for class Thread

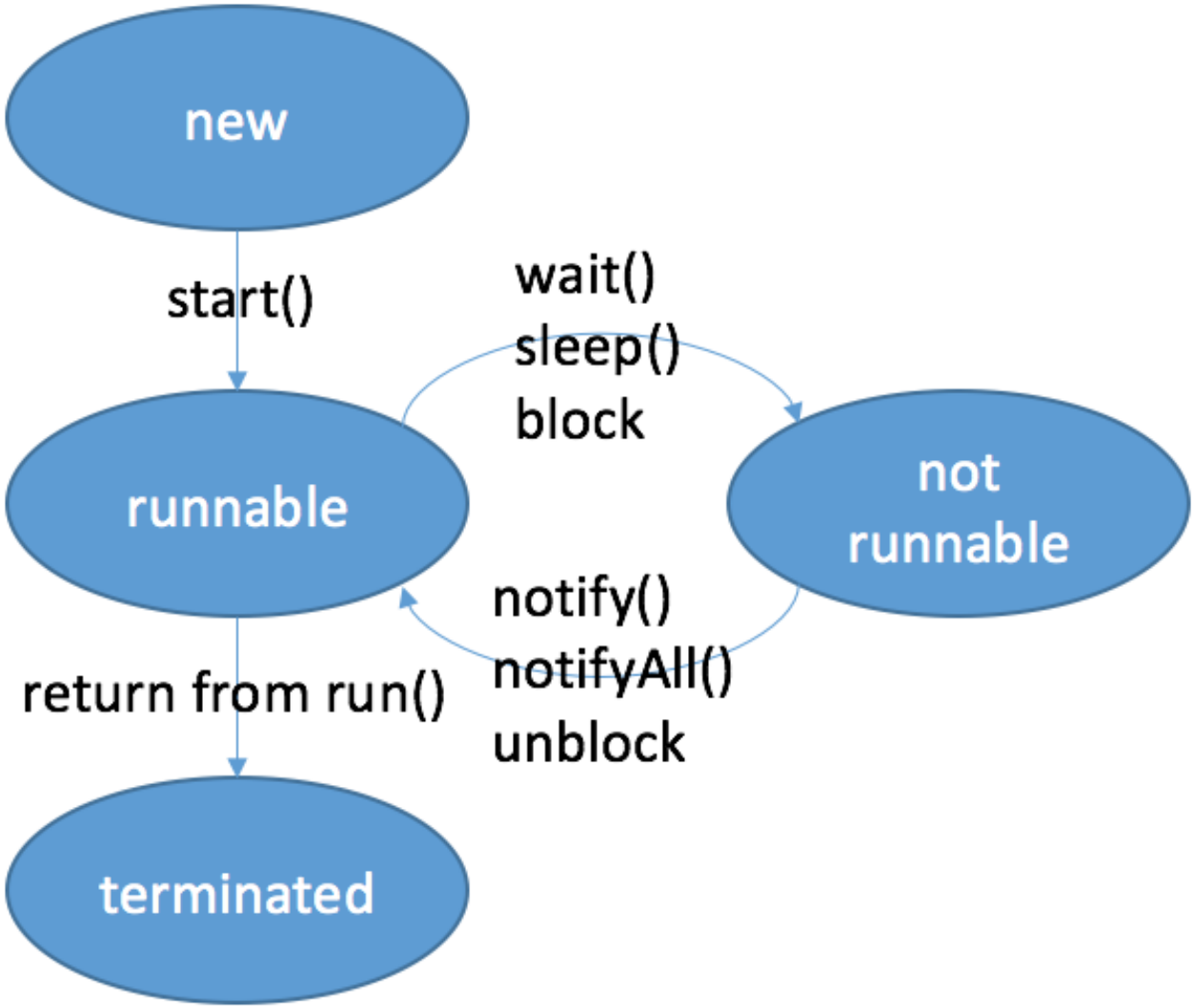
```
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
```

A thread's lifecycle

A thread has the following important states:

- new – the thread is yet to be started
- runnable – the thread is executing in the JVM
- blocked – the thread is blocked, waiting for a monitor
- timed_waiting – the thread is blocked for up to a given time
- terminated – the thread has returned from method run()

A thread's lifecycle (cont.)



This program starts 2 threads

```
public class Main {
    public static void main(String[] args) {
        new T1().start();
        new T2().start();
        System.out.println("main thread terminates");
    }
}

class T1 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++)
            System.out.println("T1: i = " + i);
        System.out.println("T1 terminates");
    }
}

class T2 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++)
            System.out.println("T2: i = " + i);
        System.out.println("T2 terminates");
    }
}
```

Non-deterministic execution (example #1)

```
T1: i = 0
T1: i = 1
T1: i = 2
T1: i = 3
T1: i = 4
T1 terminates
main thread terminates
T2: i = 0
T2: i = 1
T2: i = 2
T2: i = 3
T2: i = 4
T2 terminates
```

Non-deterministic execution (example #2)

```
T1: i = 0
T1: i = 1
T2: i = 0
main thread terminates
T2: i = 1
T2: i = 2
T2: i = 3
T2: i = 4
T2 terminates
T1: i = 2
T1: i = 3
T1: i = 4
T1 terminates
```

Invoking `sleep(long)` will stop execution for some time

- A thread calls `sleep(long millis)` to "pause" execution.
- It is a class method (use `Thread.sleep(1000)` to sleep $1s$).
- Not really suitable for synchronizing threads...

Invoking `join()` will wait for a thread to terminate

- If T1 invokes the method `join()` on T2, then T1 will block until T2 terminates.
- It works just as invoking `wait()` and checking `isAlive()`.
- Note: the `join()` caller blocks.
  - Blocking the UI thread will block the UI...
  - There is an alternative `join(long millis)` that will wait for a maximum specified time before returning.

Shared objects – using the *synchronized* keyword

- Threads (most) often share objects.
- Near-simultaneous access may lead to inconsistencies.
- Java provides the `synchronized` keyword to "synchronize" (or coordinate) access to shared data/objects.

An account shared by two threads...

```
public class Account {
    int balance;

    public Account(int balance) { this.balance = balance; }

    public void withdraw(int amount) { balance = balance - amount; }

    public int getBalance() { return balance; }

    public static void main(String[] args) {
        Account account = new Account(1000);
        new Withdraw(account).start();
        new Withdraw(account).start();
    }
}

class Withdraw extends Thread {
    Account account;

    public Withdraw(Account account) { this.account = account; }

    public void run() {
        int total = 0;
        while(account.getBalance() >= 70) {
            // try { Thread.sleep(500); } catch(InterruptedException e) {}
            account.withdraw(70);
            total = total + 70;
            System.out.println("Available: " + account.getBalance());
        }
        System.out.println("Total withdrawn: " + total);
    }
}
```

Synchronizing access to objects

Two main ways of using the synchronized keyword

- `public synchronized void withdraw(int amount) {...}`
  to guarantee that only one thread executes the method at a given time (all others are suspended)
- `synchronized(account) {...}`
  to guarantee that only one thread executes the block at a given time (all others are suspended)

Any Java object has methods `wait()` and `notify()`

- Every Java object has a *monitor* (a lock).
- An object's monitor can be owned by one thread at a time.
- Every Java object has a method to wait on the lock, and a method to notify other threads currently waiting on the lock.
- `wait()` – The current thread waits (suspended) until the notify() method is called, on the object, by some other thread.
- `notify()` – One thread currently waiting on the object's lock is awaken (unsuspended).

Simple wait/notify protocol

The wait() method must always be called in a loop:

```
synchronized (object) {
    while( condition for action does not hold )
        object.wait()
    // perform the intended action
}
```

Simple wait/notify protocol (cont.)

- The notify() method is called by the owner of an object's lock.
  - Either by invoking a synchronized method of the object,
  - or by executing a block of statements synchronized on the object.
- The awakened thread will be able to proceed only when the current owner releases the lock.

Thread-safe classes

What if 2 (or more) threads share an `ArrayList`?

- Suppose you need to maintain a *list* of users or items
- The `ArrayList` class is unsynchronized (not thread-safe)

You could write `synchronized(myArrayList){...}`

- This locks the entire list for a single thread
- Safe but slow...

Instead, you can also use a `CopyOnWriteArrayList`

- It behaves just like an `ArrayList` (same interface)
- More concurrent programming classes: `java.util.concurrent`

`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`

Bibliography

Oracle has excellent javadocs and tutorials:

- `https://docs.oracle.com/javase/7/docs/api/`
- `https://docs.oracle.com/javase/tutorial/`

Most Java books have a chapter on multithreading, such as:

- Liang, "Introduction to Java Programming, Comprehensive Version", 8th edition, Ch. 29, Pearson, 2011.

Java Threads

Raul Barbosa

Sistemas Distribuídos 2023/24