Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: concurrency

Outline

- Java threads
  - thread implementation
  - sleep, interrupt, and join
  - threads that return values
- Thread synchronization
  - implicit locks and synchronized blocks
  - synchronized methods
  - producer/consumer example
- Other concurrency models
  - executors and thread pools
  - explicit locks and semaphores
  - thread-safe collections
  - fork/join parallelism

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java threads

Java threads

Java's concurrency model is based on threads

- implemented natively in the JVM

Threads are created by instantiating class **`Thread`**

- Each instance is associated with a class providing the code associated with the thread

Two ways to provide the class code

- Write a class that implements **`interface Runnable`**
- Write a class that inherits from **`class Thread`**

We focus on the first solution, which is a bit more flexible

- Why?

Implementing **interface Runnable**

Any implementation of **Runnable** must implement method **run()**.

```
public class DumbThread implements Runnable {

    String id;

    public DumbThread(String id) {
        this.id = id;
    }

    public void run() {
            // do something when executed
            System.out.println("This is thread " + id);
    }
}
```

Starting a thread

Create a **`Thread`** object

```
Thread mt = new Thread(new DumbThread("mt"));
```

Start its execution (calls **`run()`**)

```
mt.start();
```

Optionally, wait for it to terminate

```
mt.join();  // wait until mt terminates
    System.out.println(
                    "The thread has
terminated");
```

Putting a thread to sleep

The **`sleep(int t) static`** method suspends the thread in which it is invoked for **`t`** milliseconds, or until an interrupt is received

```
Thread.sleep(2000); // suspend for 2 seconds
```

- **`sleep`** throws an **`InterruptedException`** if an interrupt occurs
- even if no interrupt occurs, the timing may be more or less precise according to the real-time guarantees of the running JVM

Threads that return values

The generic **interface Callable<G>** is a variant of **Runnable** for threads returning values of type **G**.

- must implement method **call()**

```
import java.util.concurrent.*;

public class CalThread implements Callable<String> {
	String id;
	public CalThread(String id) { this.id = id; }
	public String call() {
			return "Thread with id: " + id;
	}
}
```

Callable objects are run using executors (see later)

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Thread synchronization

Implicit locks and synchronized blocks

Synchronized blocks (a.k.a. synchronized statements) support synchronization based on locks.

- blocks of statements guarded by `synchronized`
- the lock itself can be any object (including `this`)
- locking/unlocking is implicit when entering/exiting the block
- named "intrinsic locks"
- useful to define critical regions and fine-grained synchronization

Implicit locks and synchronized blocks

Synchronized blocks (a.k.a. synchronized statements) support synchronization based on locks.

```
// s must be accessed in mutual exclusion
private int s;

// dict is used read-only, so no concurrency problems
private LinkedList<String> dict;

public String decrement_and_lookup() {
      // critical region
  synchronized(this) { if (s > 0) { s = s - 1; } }
      // non-critical region
  return dict.get(s);
}
```

Synchronized methods

Java implements monitors as **`synchronized`** methods.

When a thread is executing a **`synchronized`** method for an object, all other threads executing **`synchronized`** methods on the same object wait (i.e., they block execution).

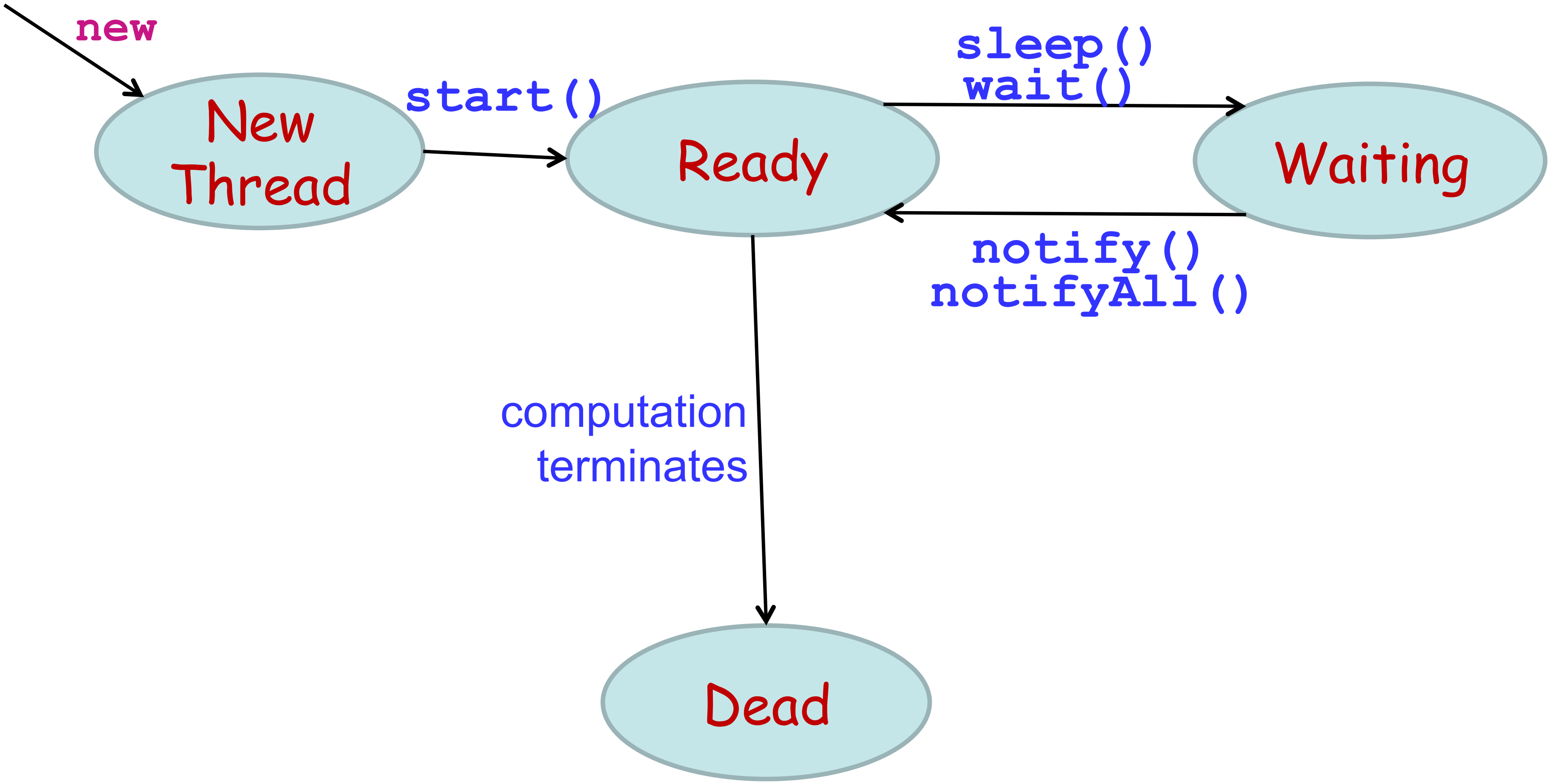
- it is as if the method acquires an implicit lock on the object and does not release it until it's done

Synchronized methods

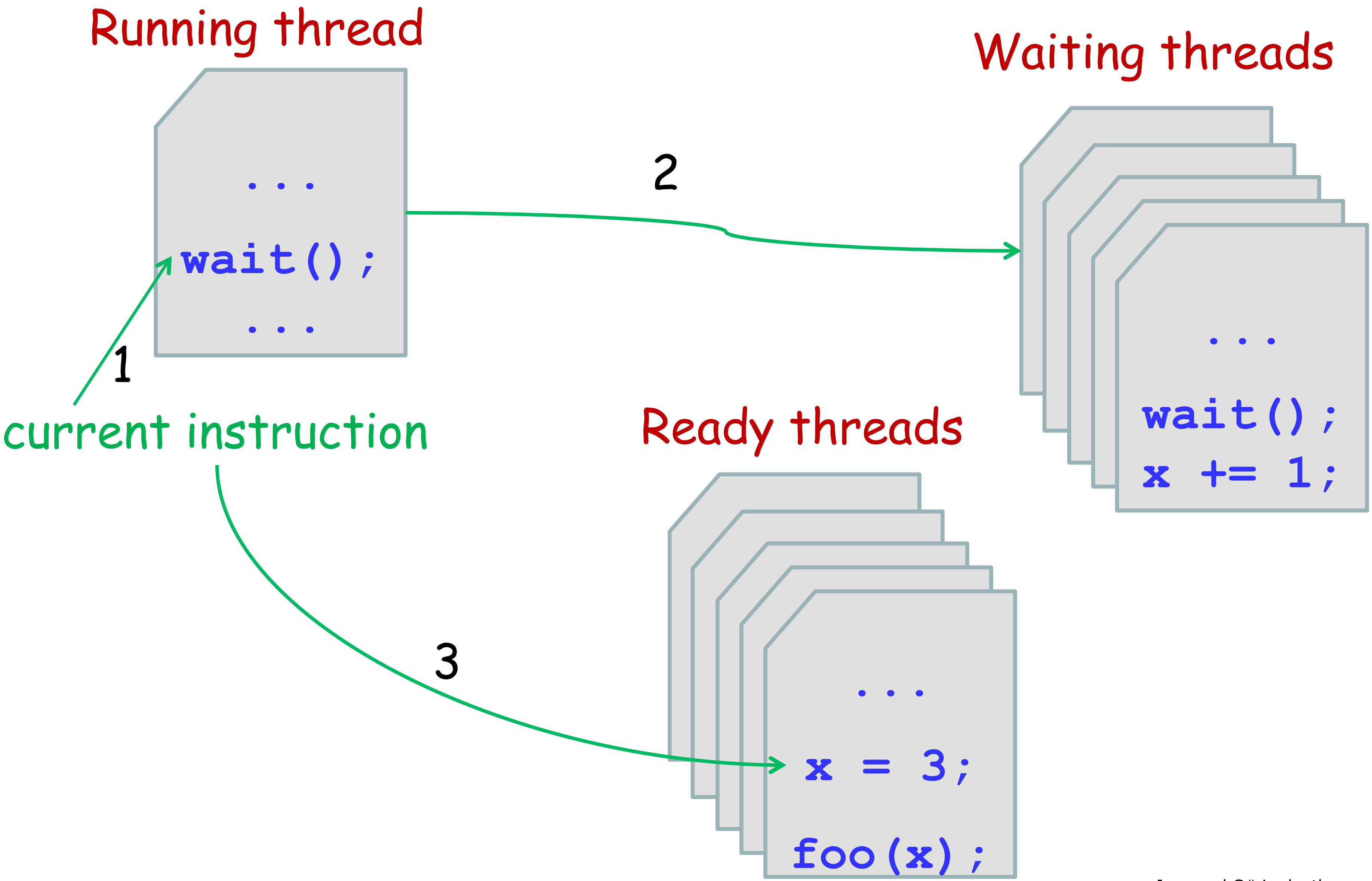
Java implements monitors as **`synchronized`** methods

- **`synchronized`** methods coordinate with the primitives
  - **`wait`**: suspend and release the lock until some thread does a **`notify`** or **`notifyAll`**
  - **`notify`**: resume one suspended thread (chosen nondeterministically), which becomes ready for execution when possible
  - **`notifyAll`**: resume all suspended threads, which become ready for execution when possible
- There is no guarantee that notifications to waiting threads are fair

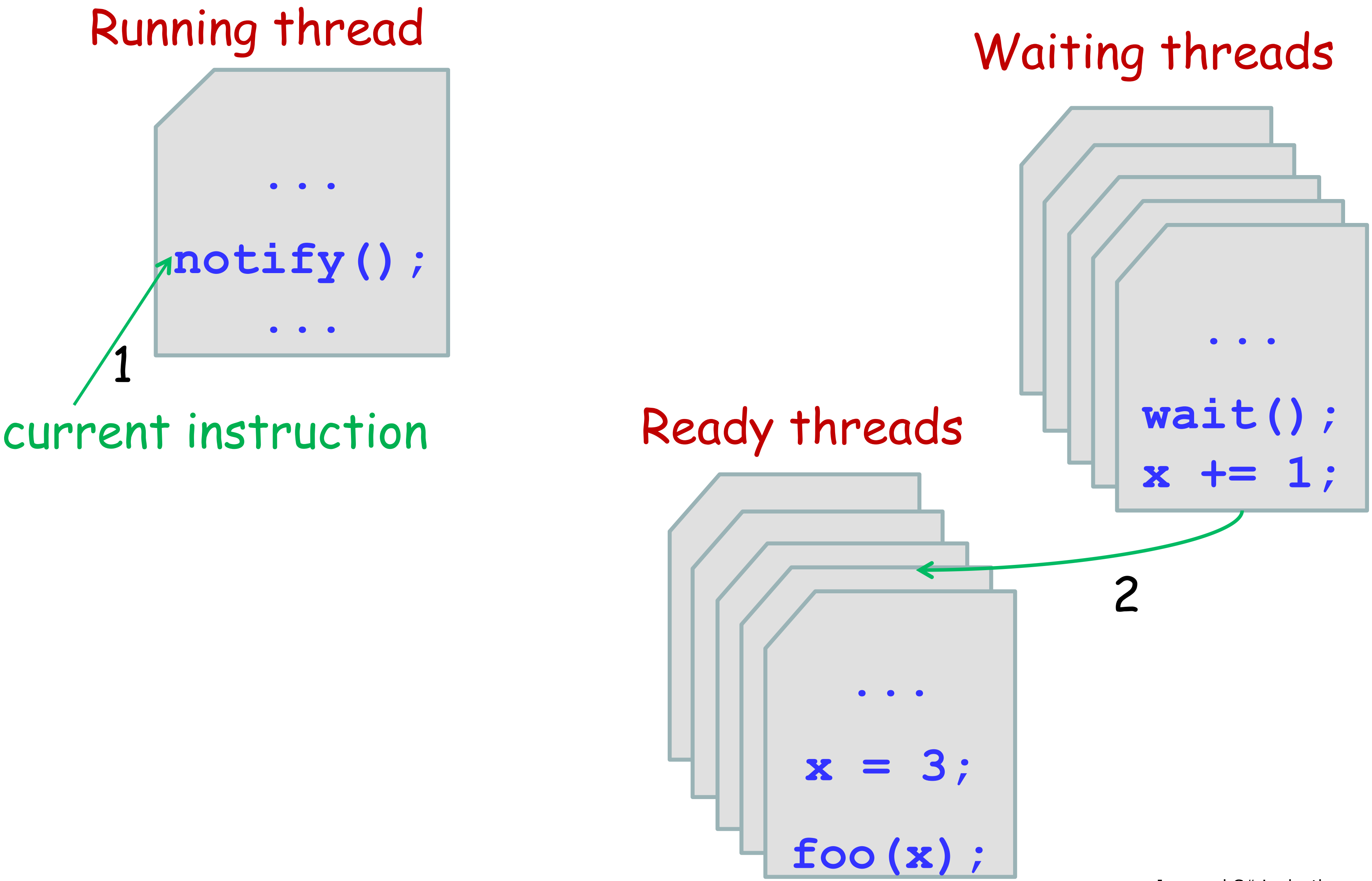
Thread states



From running to waiting



From waiting to ready



The producer-consumer problem

Two threads, the Producer and the Consumer, work concurrently on a shared Buffer of bounded size

The Producer puts new messages in the buffer

- if the buffer is full, the Producer must wait until the Consumer takes some messages
- the Producer also signals the last message

The Consumer takes messages from the buffer

- if the buffer is empty, the Consumer must wait until the Producer puts some new messages
- the Consumer terminates after the last message

Consistent access to the Buffer requires locks and synchronization

One way is to make Buffer a monitor class (with **`synchronized`** methods)

The main class

```
public class ProducerConsumer {

    public static void Main(String[] args) {
        // create a buffer of size 3
        Buffer b = new Buffer(3);
        // start the producer
        (new Thread(new Producer(b, “END”))).start();
        // start the consumer
        (new Thread(new Consumer(b, “END”))).start();
    }
}
```

The shared Buffer (1/3)

```
import java.util.*;

public class Buffer {

    public Buffer(int max_size) {
        this.max_size = max_size;
        this.messages = new LinkedList<String>();
    }

    // buffer of messages, managed as a queue
    private LinkedList<String> messages;
    // maximum number of elements in the buffer
    private int max_size;
```

The shared Buffer (2/3)

```
public synchronized String take() {
    while (messages.size() == 0) {
        wait();
        // may throw InterruptedException
    }
    // now the buffer is not empty
    // and we have exclusive access to it
    String m = messages.remove();
    // any thread waiting for a slot in the buffer
    notifyAll();
    // return the message on top of the buffer
    return m;
}
```

The shared Buffer (3/3)

```
    public synchronized void put(String msg) {
        while (messages.size() == max_size) {
            wait();
            // may throw InterruptedException
        }
        // now the buffer has at least an empty slot
        // and we have exclusive access to it
        messages.offer(msg);
        // any thread waiting for a message to take
        notifyAll();
    }

} // end of class Buffer
```

The Producer (1/2)

```
public class Producer implements Runnable {

    // reference to the shared buffer
    private Buffer b;
    // the last message to be sent
    private String endMsg;

    // set the reference to the buffer and endMsg
    public Producer(Buffer b, String endMsg) {
        this.b = b;
        this.endMsg = endMsg;
    }
```

The Producer (2/2)

```
    public void run() {

        // work for 20 turns
        for (int i = 0; i < 20; i++) {
              // put a message in the buffer
              b.put(Integer.toString(i));
        }
        // last message signals end
        b.put(endMsg);
    }
}
```

The Consumer (1/2)

```
public class Consumer implements Runnable {

    // reference to the shared buffer
    private Buffer b;
    // the last message to be sent
    private String endMsg;

    // set the reference to the buffer and endMsg
    public Consumer(Buffer b, String endMsg) {
        this.b = b;
        this.endMsg = endMsg;
    }
```

The Consumer (2/2)

```
    public void run() {
        String m = “”;  // assume endMsg != “”

        // work until endMsg is received
        for (int i = 0; !m.equals(endMsg); i++) {
              // take a message from the buffer
              m = b.take();
              System.out.println(
                  "Consumer has consumed message: " + m);
        }
    }
}
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Other concurrency models

Concurrency and performance

Thread creation is time-consuming

- massive thread creation can annihilate responsiveness
- Java's solution: executors and thread pools

Synchronized blocks and methods are not very efficient

- Java's solution
  - explicit locks (lightweight)
  - atomic variables (semaphores)

Tip: don't forget the efficiency/abstraction trade-off

Concurrency and correctness

Programming thread-safe data structures is error-prone

- Java's solution: concurrent collections

Threads and monitors are too general for straightforward parallel computation

- Java's solution: fork/join tasks

Tip: don't forget the efficiency/abstraction trade-off

Executors and thread pools

Executors are object services that run threads

Thread pools are an efficient way of implementing executors

- maintain a pool of worker threads
- when a client requests a new task to run, preempt one of the available worker threads and assign it to the task
- no creation overhead upon task invocation

Java has three interfaces for the services of an executor

- `Executor`: defines `executor` method for `Runnable` objects
- `ExecutorService`: supports `Runnable` and `Callable` objects
- `ScheduledExecutorService`: supports scheduled execution (at a given time)

Class `java.util.concurrent.Executors`

This class is an object factory for several efficient implementations of executors (mostly with thread pools)

- **`newFixedThreadPool`**: returns an executor that uses a thread pool of fixed size
- **`newCachedThreadPool`**: returns an executor that uses a thread pool of variable size
- use **`submit`** to send **`Runnable`** or **`Callable`** objects to be executed

Class Executors also provide implementations of **`ScheduledExecutionService`**

Executors vs. standard thread creation

Without executors

```
Thread t1 = new
      Thread(new T_a());
Thread t2 = new
      Thread(new T_a());
Thread t3 = new
      Thread(new T_b());

t1.start();
t2.start();
t3.start();
```

With executors

```
ExecutorService e =
   Executors.newCachedThre
   adPool();

t1 = new T_a();
t2 = new T_a();
t3 = new T_b();

e.submit(t1);
e.submit(t2);
e.submit(t3);
```

Executing a **`Callable`** object

- You can submit a **`Callable`** object to an executor
- The executor returns a **`Future`** object, used to read the result of the execution returned by the thread

```
ExecutorService e = new CachedThreadPool();

Callable<G> t = new aCallableClassReturningG();
Future<G> f = e.submit(t);
G result = f.get();  // may throw an exception
```

Explicit locks

Java locks in package **`java.util.concurrent.lock`** provide:

- explicit locking mechanisms:
  - **`lock`**: acquire the lock if available, and wait until it becomes available otherwise
  - **`lockInterruptibly`**: try to lock, but waiting can be interrupted
  - **`tryLock`**:
    - if lock available, acquire it immediately and return true
    - if lock not available, return false (and don't wait)
  - **`unlock`**: release the lock
- more complex reentrant locking mechanisms
  - wait for a specific signal or condition
  - query the lock to know how many threads are waiting
  - …

Atomic variables

Java's implementation of semaphore-like objects

- in `java.util.concurrent.atomic`

```
// shared variable, initialized to 0
AtomicInteger s = new AtomicInteger(0);

...
// this is equivalent to an atomic s++
s.incrementAndGet();
...
// this is equivalent to an atomic s--
s.decrementAndGet();
```

Concurrent collections

Java provides several implementations of data structures that are thread-safe

- `LinkedBlockingQueue`
- `ArrayBlockingQueue`
- `ConcurrentHashMap`
- …

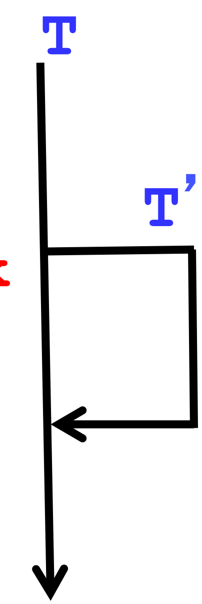
A thread-safe list implementation is also provided among the standard collections in `java.utils.Collections`

```
public static List synchronizedList(List list)
```

Fork/join parallelism

Fork/join is a straightforward model of parallel computation suitable to implement divide and conquer algorithms exploiting parallelism.

```
x = instance to be solved;
if ( x is small ) {
        solve x;
} else {
        split x into x1 and x2;
        spawn a new thread T' and launch it on x1;  // fork
        recursively solve x2;
        wait until T' is done;                      // join
        combine the solutions for x1 and x2
            into a solution for x;
}
```



Fork/join parallelism

Java 7 introduced a library for fork/join parallelism (in `java.util.concurrent`).

`ForkJoinPool` is a specialized executor service, which handles tasks that can fork and join. Its main purpose is making sure that no thread is idle ("work stealing" schedule).

`RecursiveAction` and `RecursiveTask<T>` are the two main abstract classes to define tasks that can fork and join.

- `RecursiveAction` for tasks that don't return any value.
- `RecursiveTask<T>` for tasks that return values of type `T`.
- Inherit and override `T compute()` to implement specific tasks (`T` is `void` for `RecursiveAction`).

Fork/join parallelism

Main methods of `RecursiveAction` and `RecursiveTask<T>` (`T` is `void` for `RecursiveAction`):

- `fork()`: schedule task for asynchronous parallel execution.
- `T join()`: await for task termination and return result.
- `T invoke()`: arrange parallel execution, await for termination, and return result.
- `invokeAll(Collection<T> tasks)`: spawn multiple tasks and wait for all of them to terminate (works on tasks in the collection passed as argument).

Fork/join parallelism: example

Divide and conquer algorithm to sum the content of an array:

1. If the array is small, iterate over its values.
2. Otherwise, split it in two, sum the two halves in parallel, and then combine the two partial sums.

```
public class ParSum extends RecursiveTask<Integer> {
   int [] values;  // values to be summed
   int low, high;  // range to be summed

   public ParSum (int [] values, int low, int high) {
      this.values = values;
      this.low = low;   this.high = high;
   }
```

Fork/join parallelism: example (cont'd)

```
public class ParSum extends RecursiveTask<Integer> {
    int [] values;  // values to be summed
    int low, high;  // range to be summed

        // is the range "small"?
    protected boolean isSmall() {
        return (high - low + 1 < 4);
    }

        // compute sum directly
    protected int computeDirectly() {
        int sum = 0;
        for (int i = low, i <= high; i++)
            sum += values[i];
        return sum;
    }
```

Fork/join parallelism: example (cont'd)

```
@Override
protected Integer compute() {
    if ( isSmall() ) {
            // directly compute small instances
        return computeDirectly();
    } else {
            // split into two halves
            // note: what's wrong with (low+high)/2?
        int mid = low + (high - low + 1)/2;
        ParSum t1 = new ParSum(values, low, mid);
        t1.fork();  // fork a thread on lower half
        low = mid + 1; // current thread on upper half
            // overall result: upper sum + lower sum
        return compute() + t1.join();
    }
}
```

Fork/join parallelism: example (cont’d)

How to start the parallel computation and get the result:

```
int [] data = ... ; // get data, somehow
ParSum sum = new ParSum(data, 0, data.length-1);
ForkJoinPool pool = new ForkJoinPool();
int total = pool.invoke(sum);
System.out.println (“The sum is ” + total);
```