Outline

Outline

Processes and Threads

- **process**: a running program
  - each process has its exclusive memory
  - managed by the operating system, heavy weight
  - **distributed computing**: running on different machines
  - may have several threads
- **multitasking**: run several **processes** in parallel
  - OS switches between processes
  - **multiprocessing**: run in parallel on several processors

- **thread**: a running subprogram
  - threads can share memory (**shared address space**)
  - managed by the program, light weight
- **multithreading**: run several **threads** in parallel
  - may be switched between processors or run on several in parallel

Threads APIs

- POSIX threads (Pthreads; IEEE standard 1995): C
- C++ Standard library thread (2011): C++
- Java Standard library (1995, 2004): Java
  - Thread (java.lang; 1995)
  - Lock, ThreadPoolExecutor etc. (java.util.concurrent; 2004)
- Open Multi-Processing (OpenMP; 1997): C, C++, Fortran
  - Java ports: JOMP, omp4j

Outline

Starting Threads

Runnable:

- interface in java.lang
- only method: **void run()**.
- models a procedure that can be run.

Thread:

- class in java.lang
- constructor **Thread(Runnable)**: a thread to run a given Runnable.
- **Thread.start()**: begin to execute this thread.

Starting Threads / Example

```
1  public class HelloWorld3 implements Runnable {
2      String msg;
3      public HelloWorld3(String msg) {
4          this.msg = msg;
5      }
6      public void run() {
7          while (true)
8              System.out.println(msg);
9      }
10     public static void main(String[] args) {
11         new Thread(new HelloWorld3("A")).start();
12         new Thread(new HelloWorld3("B")).start();
13         new Thread(new HelloWorld3("C")).start();
14         new Thread(new HelloWorld3("D")).start();
15     }
16 }
```

Output

```
A
C
B
A
A
D
.
.
.
```


Interrupting Threads

- **Thread.interrupt()**:
  - set the thread's interrupted property to true.
  - if the thread is sleeping or waiting, an **InterruptedException** will be thrown.
- **Thread.interrupted()**:
  - get the value of the thread's interrupted property.
- **Thread.isAlive()**:
  - a thread is alive if it has been started, but not yet died.
- **Thread.currentThread()** (static):
  - get the thread executing the current code.
- **Thread.sleep(long)** (static):
  - the current thread sleeps for the given number of miliseconds.
- There is no way to stop a thread externally.
- There is no way to interrupt a thread that does not cooperate.

Interrupting Threads / Example / Example Computation

```
1  import java.util.*;
2
3  public class Primes {
4      ArrayList<Long> primes = new ArrayList<>();
5
6      public void compute(long max) {
7          primes.add(2L);
8          for (long n = 3; n < max; n = n+2) {
9              boolean isPrime = true;
10             for(Long prime: primes)
11                 if (n % prime == 0) {
12                     isPrime = false;
13                     break;
14                 }
15             if (isPrime)
16                 primes.add(n);
17         }
18     }
19 }
```


Interrupting Threads / Example (non coop./broken)

```
1  import java.util.*;
2
3  public class Worker implements Runnable {
4      public void run() {
5          for (int i = 0; i < 1000; ++i) {
6              System.out.println("Work␣step␣" + i);
7              new Primes().compute(100000);
8          }
9      }
10
11     public static void main(String[] args) {
12         Thread worker = new Thread(new Worker());
13         worker.start();
14         while (worker.isAlive()) {
15             String input = System.console().readLine();
16             if (input.equals("interrupt")) {
17                 worker.interrupt();
18                 break;
19             }
20         }
21     }
22 }
```

Output

```
Work step 0
Work step 1
Work step 2
interrupt
Work step 3
.
.
.
```

Interrupting Threads / Example (coop./fixed)

```
1  import java.util.*;
2
3  public class Worker2 implements Runnable {
4      public void run() {
5          for (int i = 0; i < 1000; ++i) {
6              if (Thread.currentThread().isInterrupted())
7                  break;
8              System.out.println("Work␣step␣" + i);
9              new Primes().compute(100000);
10         }
11     }
12
13     public static void main(String[] args) {
14         Thread worker = new Thread(new Worker2());
15         worker.start();
16         while (worker.isAlive()) {
17             String input = System.console().readLine();
18             if (input.equals("interrupt")) {
19                 worker.interrupt();
20                 break;
21             }
22         }
23     }
24 }
```

Output

```
Work step 0
Work step 1
Work step 2
interrupt
```


Interrupting Threads / Example (sleeping)

```
1  import java.util.*;
2
3  public class Worker3 implements Runnable {
4      public void run() {
5          for (int i = 0; i < 1000; ++i) {
6              System.out.println("Work␣step␣" + i);
7              try {
8                  Thread.sleep(1000);
9              } catch (InterruptedException ex) {
10                 break;
11             }
12         }
13     }
14
15     public static void main(String[] args) {
16         Thread worker = new Thread(new Worker3());
17         worker.start();
18         while (worker.isAlive()) {
19             String input = System.console().readLine();
20             if (input.equals("interrupt")) {
21                 worker.interrupt();
22                 break;
23             }
24         }
25     }
26 }
```

Outline

Synchronization

- Several threads running in parallel may need to exchange some information.
  - can be accomplished through shared variables
- Several threads running in parallel may need to coordinate, e.g.,
  - a thread needs to wait until another is terminated
  - a thread requires exclusive access to some variable
    - e.g., to increment a counter or to edit an array
  - a thread requires some condition to hold to continue
    - e.g., further input in a stream being available
- Called **synchronization**.

Waiting for Termination

- **Thread.join()**:
  - the current thread sleeps until the target thread dies.
- When a program is started, there exists one thread (often called main thread).
- a program terminates once all its threads died.

Waiting for Termination / Example

```
1  public class Counter {
2      int count = 0;
3      public void increment() { ++count; }
4      public int value() { return count; }
5  }
```

```
1  public class ParallelCounters implements Runnable {
2      Counter count;
3      int num;
4
5      public ParallelCounters(Counter count, int num) {
6          this.count = count;
7          this.num = num;
8      }
9      public void run() {
10         for (int i = 0; i < num; ++i)
11             count.increment();
12     }
13
14     public static void main(String[] args) throws InterruptedException {
15         Counter count = new Counter();
16         Thread a = new Thread(new ParallelCounters(count, 100));
17         Thread b = new Thread(new ParallelCounters(count, 100));
18         Thread c = new Thread(new ParallelCounters(count, 100));
19         Thread d = new Thread(new ParallelCounters(count, 100));
20
21         a.start(); b.start(); c.start(); d.start();
22
23         System.out.println("counter:␣" + count.value());
24     }
25 }
```

Waiting for Termination / Example

```
1  public class ParallelCounters2 implements Runnable {
2      Counter count;
3      int num;
4
5      public ParallelCounters2(Counter count, int num) {
6          this.count = count;
7          this.num = num;
8      }
9      public void run() {
10         for (int i = 0; i < num; ++i)
11             count.increment();
12     }
13
14     public static void main(String[] args) throws InterruptedException {
15         Counter count = new Counter();
16         Thread a = new Thread(new ParallelCounters2(count, 100));
17         Thread b = new Thread(new ParallelCounters2(count, 100));
18         Thread c = new Thread(new ParallelCounters2(count, 100));
19         Thread d = new Thread(new ParallelCounters2(count, 100));
20
21         a.start(); b.start(); c.start(); d.start();
22
23         a.join(); b.join(); c.join(); d.join();
24
25         System.out.println("counter:␣" + count.value());
26     }
27 }
```


Synchronized Methods

- Even steps of very simple statements such as **++value** may be interleaved with steps in other threads and lead to corruption.
- For each object and class there exists an implicit lock (called monitor).
- Methods marked **synchronized**
  - try to acquire the monitor of their object and
  - block if the monitor is already taken by another thread until it becomes available.
  - thus, there is at most one thread executing any synchronized method at any time.
- **static synchronized** methods try to acquire the monitor of the class.
- The **synchronized(Object) { . . . }** statement tries to acquire the monitor of the given object/class.
- **Thread.holdsLock(Object)** (static) tests if the current thread holds a given monitor.

Synchronized Methods / Example

```
1  public class Counter2 {
2      int count = 0;
3      public synchronized void increment() { ++count; }
4      public int value() { return count; }
5  }
```

```
1  public class ParallelCounters3 implements Runnable {
2      Counter2 count;
3      int num;
4
5      public ParallelCounters3(Counter2 count, int num) {
6          this.count = count;
7          this.num = num;
8      }
9      public void run() {
10         for (int i = 0; i < num; ++i)
11             count.increment();
12     }
13
14     public static void main(String[] args) throws InterruptedException {
15         Counter2 count = new Counter2();
16         Thread a = new Thread(new ParallelCounters3(count, 100));
17         Thread b = new Thread(new ParallelCounters3(count, 100));
18         Thread c = new Thread(new ParallelCounters3(count, 100));
19         Thread d = new Thread(new ParallelCounters3(count, 100));
20
21         a.start(); b.start(); c.start(); d.start();
22
23         a.join(); b.join(); c.join(); d.join();
24
25         System.out.println("counter:␣" + count.value());
```


Synchronization Issues

- Deadlock:
  - Thread A is waiting for Thread B,
    Thread B is waiting for Thread A.
  - Thread A holds lock 1 and requests lock 2,
    Thread B holds lock 2 and requests lock 1.
  - The program freezes.

Synchronization Issues / Deadlock / Example

```
1  class Account {
2      String id;
3      double balance = 0;
4
5      Account(String id) { this.id = id; }
6
7      void withdraw(double amount) { balance -= amount; }
8      void deposit(double amount) { balance += amount; }
9
10     static void transfer(Account from, Account to, double amount) {
11         synchronized(from) {
12             synchronized(to) {
13                 from.withdraw(amount);
14                 to.deposit(amount);
15             }
16         }
17     }
18 }
```


Synchronization Issues / Deadlock / Example (ctd.)

```
1  class ParallelTransactions implements Runnable {
2      Account from, to;
3      ParallelTransactions(Account from, Account to) {
4          this.from = from;
5          this.to = to;
6      }
7      public void run() {
8          while (true) {
9              Account.transfer(from, to, 100.00);
10             System.out.println("transfered␣100.00␣from␣" + from.id + "␣to␣" + to.id);
11             try {
12                 Thread.sleep(1000);
13             } catch (InterruptedException ex) {
14                 break;
15             }
16         }
17     }
18     public static void main(String[] args) {
19         Account a = new Account("A"), b = new Account("B");
20         new Thread(new ParallelTransactions(a, b)).start();
21         new Thread(new ParallelTransactions(b, a)).start();
22     }
23 }
```

Synchronization Issues / Deadlock / Example (fix)

```
1  class Account2 {
2      String id;
3      double balance = 0;
4
5      Account2(String id) { this.id = id; }
6
7      void withdraw(double amount) { balance -= amount; }
8      void deposit(double amount) { balance += amount; }
9
10     static synchronized void transfer(Account2 from, Account2 to, double amount) {
11         synchronized(from) {
12             synchronized(to) {
13                 from.withdraw(amount);
14                 to.deposit(amount);
15             }
16         }
17     }
18 }
```


Conditions / Guarded Blocks

- Often threads require a specific condition to hold before they can resume their work.
- **polling**:
  - repeatedly query the condition, proceed if it holds
  - wastes resources
  - possibly sleep between trials
    - sleep time not straight-forward to set
- Condition:
  - a queue of threads to wait for a condition to become true
  - a method to wait on such a condition (**Object.wait**)
  - a method to signal that the condition may have changed (**Object.notifyAll**)
    - The condition itself is not part of the model.

Guarded Blocks / Example (1/2)

```
1  import java.util.*;
2  class Store {
3      ArrayList<String> store = new ArrayList<>();
4      synchronized void put(String item) { store.add(item); }
5      synchronized String pop() { String item = store.get(0); store.remove(0); return item; }
6  }
```

```
1  class Producer implements Runnable {
2      Store store;
3      public Producer(Store store) { this.store = store; }
4      public void run() {
5          while (true) {
6              try {
7                  Thread.sleep(Math.round(Math.random() * 1000));
8              } catch (InterruptedException ex) {}
9              String item = "A";
10             store.put(item);
11             System.out.println("produced␣" + item + ",␣store␣=␣" + store.store);
12         }
13     }
14 }
```


Guarded Blocks / Example (2/2)

```
1  class Consumer implements Runnable {
2      Store store;
3      public Consumer(Store store) { this.store = store; }
4      public void run() {
5          while (true) {
6              if (store.store.size() >= 2) {
7                  String item1 = store.pop(), item2 = store.pop();
8                  System.out.println("consumed␣" + item1 + "␣and␣" + item2);
9                  try {
10                     Thread.sleep(Math.round(Math.random() * 1000));
11                 } catch (InterruptedException ex) {}
12             }
13         }
14     }
15 }
```

Output

```
produced A, store = [A]
produced A, store = [A, A]
produced A, store = [A, A, A]
produced A, store = [A, A, A, A
produced A, store = [A, A, A, A
⋮
```

```
1  class PCExample {
2      public static void main(String[] args) {
3          Store store = new Store();
4          Producer prod = new Producer(store);
5          Consumer cons = new Consumer(store);
6          new Thread(prod).start();
7          new Thread(cons).start();
8      }
9  }
```

Guarded Blocks / Example (fix; 1/2)

```
1  import java.util.*;
2  class Store {
3      ArrayList<String> store = new ArrayList<>();
4      public synchronized void put(String item) {
5          synchronized(store) {
6              store.add(item);
7          }
8          notifyAll();
9      }
10     public String pop() {
11         String item;
12         synchronized(store) {
13             item = store.get(0);
14             store.remove(0);
15         }
16         return item;
17     }
18     public String toString() {
19         String s;
20         synchronized(store) {
21             s = store.toString();
22         }
23         return s;
24     }
25 }
```

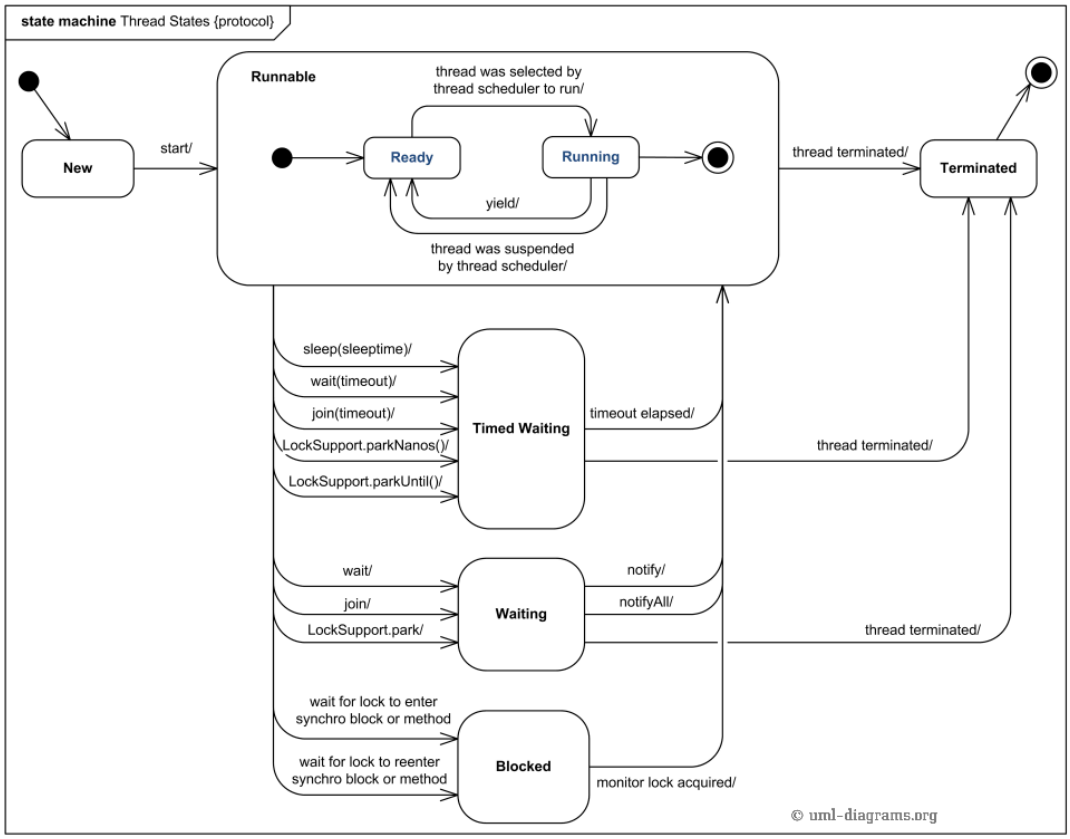

Guarded Blocks / Example (fix; 2/2)

```
1  class Consumer implements Runnable {
2      Store store;
3      public Consumer(Store store) { this.store = store; }
4      public void run() {
5          try {
6              while (true) {
7                  if (store.store.size() >= 2) {
8                      String item1 = store.pop(), item2 = store.pop();
9                      System.out.println("consumed␣" + item1 + "␣and␣" + item2
10                     Thread.sleep(Math.round(Math.random() * 1000));
11                 } else
12                     synchronized(store) {
13                         store.wait();
14                     }
15             }
16         } catch (InterruptedException ex) {}
17     }
18 }
```

Output

```
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
produced A, store = [A, A]
consumed A and A, store = []
produced A, store = [A]
```

Thread States



Information about threads at runtime

- **Thread.currentThread** (static): thread executing current code.
- **Thread.getState**: state of the thread.
- **Thread.getId**: get a numeric ID of the thread.
- **Thread.getActiveCount**: get number of concurrent threads.
- **Thread.enumerate**: get all concurrent threads.
- **Thread.getThreadGroup**: get group of the thread.

Outline

Locks

- A lock models mutually exclusive access to a resource.
  - only one thread can hold a lock at any time.
  - locks have methods to acquire and release them.
- **ReentrantrantLock**: reentrant implementation of interface **Lock**
  - reentrant: bookkeeping for repeated acquisitions and releases by the same thread.
- **Lock.lock**: acquire the lock, if possible, block otherwise until it becomes available.
- **Lock.unlock**: release the lock.
- **Lock.tryLock**: acquire the lock if possible, do nothing otherwise, return success.
  - atomic method for **Thread.holdsLock** followed by **synchronized**.
- in package **java.util.concurrent.locks**

Locks / Example

```
1  import java.util.concurrent.locks.*;
2
3  class Account3 {
4      String id;
5      double balance = 0;
6      private final Lock lock = new ReentrantLock();
7
8      Account3(String id) { this.id = id; }
9
10     void withdraw(double amount) { balance -= amount; }
11     void deposit(double amount) { balance += amount; }
12
13     static boolean transfer(Account3 from, Account3 to, double amount) {
14         boolean from_lock = from.lock.tryLock();
15         boolean to_lock = to.lock.tryLock();
16         if (from_lock && to_lock) {
17             from.withdraw(amount);
18             to.deposit(amount);
19         }
20         if (from_lock)
21             from.lock.unlock();
22         if (to_lock)
23             to.lock.unlock();
24         return from_lock && to_lock;
25     }
26 }
```


Locks / Example

```
1  class ParallelTransactions3 implements Runnable {
2      Account3 from, to;
3      ParallelTransactions3(Account3 from, Account3 to) {
4          this.from = from;
5          this.to = to;
6      }
7      public void run() {
8          while (true) {
9              while (! Account3.transfer(from, to, 100.00)) {
10                 System.out.println("accounts busy, delay somewhat");
11                 try {
12                     Thread.sleep(1000);
13                 } catch (InterruptedException ex) {}
14             }
15             System.out.println("transfered 100.00 from " + from.id + " to " +
16             try {
17                 Thread.sleep(1000);
18             } catch (InterruptedException ex) {}
19         }
20     }
21     public static void main(String[] args) {
22         Account3 a = new Account3("A"), b = new Account3("B");
23         new Thread(new ParallelTransactions3(a, b)).start();
24         new Thread(new ParallelTransactions3(b, a)).start();
25     }
26 }
```

Output

```
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
transfered 100.00 from B to A
accounts busy, delay somewhat
transfered 100.00 from A to B
transfered 100.00 from B to A
transfered 100.00 from A to B
accounts busy, delay somewhat
transfered 100.00 from B to A
```

Locks / Good Practice

- if an exception is thrown after **Lock.lock**,
  in simple sequential code **Lock.unlock** may never be executed.

```
1  lck.lock();
2  ... // do something that may throw an exception
3  lck.unlock();
```

- better wrap into a **try–finally** block:

```
1  try {
2    lck.lock();
3    ... // do something that may throw an exception
4  } finally {
5    lck.unlock();
6  }
```


Outline

Thread pools

- Avoid creation and destruction of thread objects.
- Recycle thread objects, assigning Runnables to instances from a pool.

- **ExecutorService** (interface):
  - **submit(Runnable)**: execute a runnable.
  - **shutdown**: wait for all submitted threads to complete.

- **Executors.newFixedThreadPool(int)** (static):
  - create an **ExecutorService** with a fixed number of threads
    - never run more than given number of threads in parallel.

Thread pools / Example

```
1  import java.util.concurrent.*;
2
3  public class ExThreadpool implements Runnable {
4      String name;
5      public ExThreadpool(String name) { this.name = name; }
6
7      public void run() {
8          System.out.println("start␣" + name);
9          try {
10             Thread.sleep(1000);
11         } catch (InterruptedException ex) {}
12         System.out.println("end␣" + name);
13     }
14
15     public static void main(String[] args) {
16         int cores = Runtime.getRuntime().availableProcessors();
17         System.out.println("#cores␣=␣" + cores);
18
19         ExecutorService pool = Executors.newFixedThreadPool(cores);
20         for (int i = 0; i < 2*cores; ++i)
21         pool.submit(new ExThreadpool("" + i));
22             // pool.execute(new ExThreadpool("" + i));
23         pool.shutdown();
24     }
25 }
```

Output

```
cores = 4
start 0
start 1
start 2
start 3
end 0
end 1
start 4
start 5
end 2
start 6
end 3
start 7
end 4
end 5
end 6
end 7
```

Dependency Graphs

- computation composed of several atomic parts: **tasks**
- some tasks require the results of others as input
  - ⇝ dependency graph
- encapsulate access to such results in an object: **Future<T>**
  - **Future<T>.get()**:
    - wait until the producing task is completed
    - then return the result
    - throw an **ExecutionException** if anything goes wrong
- abstract functions as interface: **Callable<T>**
  - like **Runnable**, but
  - returns a **Future<T>** – a function, not a procedure
  - may throw exceptions.
- **ExecutorService.submit(Callable<T>)**: execute a callable.

Dependency Graphs / Example

```
1  import java.util.concurrent.*;
2
3  public class ExFuture2 {
4      public static class Constant implements Callable<Double> {
5          Double value;
6          public Constant(Double value) { this.value = value; }
7          public Double call() throws InterruptedException {
8              System.out.println("Start␣computing␣constant");
9              Thread.sleep(Math.round(value * 100));
10             System.out.println("Compute␣constant␣" + value);
11             return value;
12         }
13     }
14     public static class Sum implements Callable<Double> {
15         Future<Double> d1, d2;
16         public Sum(Future<Double> d1, Future<Double> d2) {
17             this.d1 = d1; this.d2 = d2;
18         }
19         public Double call() throws InterruptedException, ExecutionException {
20             System.out.println("Start␣computing␣sum");
21             Double v1 = d1.get(), v2 = d2.get();
22             Thread.sleep(1000);
23             System.out.println("Compute␣sum␣" + v1 + "␣+␣" + v2 + "␣=␣" + (v1+v2));
24             return v1 + v2;
25         }
26     }
```

Dependency Graphs / Example

```
27     public static void main(String[] args) throws InterruptedException, ExecutionException {
28         ExecutorService pool = Executors.newFixedThreadPool(8);
29         Future<Double> c3_res = pool.submit(new Constant(3.0)),
30             c5_res = pool.submit(new Constant(5.0)),
31             c6_res = pool.submit(new Constant(6.0)),
32             sum1_res = pool.submit(new Sum(c5_res, c6_res)),
33             sum2_res = pool.submit(new Sum(c3_res, c5_res)),
34             sum3_res = pool.submit(new Sum(sum1_res, sum2_res));
35         System.out.println("(3+5)+(5+6)␣=␣" + sum3_res.get());
36         pool.shutdown();
37     }
38 }
```

Output

```
Start computing constant
Start computing constant
Start computing constant
Start computing sum
Start computing sum
Start computing sum
Compute constant 3.0
Compute constant 5.0
Compute constant 6.0
Compute sum 3.0 + 5.0 = 8.0
Compute sum 5.0 + 6.0 = 11.0
Compute sum 11.0 + 8.0 = 19.0
(3+5)+(5+6) = 19.0
```


Further Thread Classes

- concurrent collections:
  - provide atomic thread-safe query and edit operations for collections

Outline

Open MP

- Open Multi-Processing (OpenMP; 1997): C, C++, Fortran
  - Java ports:
    - JOMP: seems no longer available?
    - JAMP:
    - omp4j
- Multithreading directives are added as comments to the code.
  - starting with **omp**
- Special preprocessor **omp4j**:
  - Replace comments by code using the Java threads API.
  - Then compile the code using the standard compiler.

Parallel Sections / Example

Output

```
hello hello hello hello
```

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3     // omp parallel threadNum(4)
4     {
5             System.out.print("hello␣");
6         }
7     }
8 }
```


Parallel Sections / Example / Under the Hood

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         class OMPContext {}
4         final OMPContext ompContext = new OMPContext();
5         final org.omp4j.runtime.IOMPExecutor ompExecutor = new org.omp4j.runtime.DynamicExecutor(4);
6
7         for (int ompI = 0; ompI < 4; ompI++) {
8             ompExecutor.execute(new Runnable(){
9                     @Override
10                    public void run() {
11                            System.out.print("hello␣");
12                    }
13                });
14        }
15        ompExecutor.waitForExecution();
16    }
17 }
```

Parallel For / Example

Output

```
0213458679
```

```
1 public class ExParallelFor {
2     public static void main(String[] args) {
3         // omp parallel for
4         for (int i = 0; i < 10; i++) {
5             System.out.print(i);
6         }
7     }
8 }
```


Directives

Directives:

Directive	Usage	Behavior
// omp parallel	Before {...}	The statement will be invoked in parallel (as many threads as possible).
// omp parallel for	Before for-loop	The for-loop will be iterated in parallel.
// omp sections	Before {...}	Wrapper for // omp section directives. It may not contain any other code
// omp section	Before {...}	The statement will be invoked together with other sections in parallel.
// omp critical	Before {...}	At most one thread will access the statement at any particular time.
// omp barrier	Before {}	All threads stop here until the for the last one.
// omp master	Before {...}	Only master thread will execute the statement.
// omp single	Before {...}	Only one thread will execute the statement, no matter which one.

Attributes:

Attribute	Behavior
threadNum(N)	The directive will be invoked with N threads. Default value is set to number of CPUs.
schedule(dynamic|static)	The directive will use dynamic or static executor. Default value is set to dynamic.
public(a,b)	Variables a and b are shared among all threads.
private(a,b)	Variables a and b are created (via parameter-less constructor) for each thread separately.
firstprivate(a,b)	Variables a and b are created (via copy-constructor) for each thread separately.

Outline

Primes / Sequential

```
1  import java.util.*;
2
3  public class Eratosthenes {
4      boolean[] is_prime;
5      public void compute(int max) {
6          is_prime = new boolean[max+1];
7          for (int i = 1; i <= max; ++i)
8              is_prime[i] = true;
9          for (int i = 2; i < Math.floor(Math.sqrt(max)); ++i) {
10             if (is_prime[i]) {
11                 for (int j = 2*i; j <= max; j += i) {
12                     is_prime[j] = false;
13                 }
14             }
15         }
16     }
17     public static void main(String[] args) {
18         Eratosthenes primes = new Eratosthenes();
19         primes.compute(1000000000);
20     }
21 }
```

Primes / Bad Parallelization

```
1  import java.util.*;
2
3  public class Eratosthenes2 {
4      boolean[] is_prime;
5      public void compute(int max) {
6          is_prime = new boolean[max+1];
7          for (int i = 1; i <= max; ++i)
8              is_prime[i] = true;
9          for (int i = 2; i < Math.floor(Math.sqrt(max)); ++i) {
10             if (is_prime[i]) {
11                 // omp parallel for
12                 for (int j = 2*i; j <= max; j += i) {
13                     is_prime[j] = false;
14                 }
15             }
16         }
17     }
18     public static void main(String[] args) {
19         Eratosthenes2 primes = new Eratosthenes2();
20         primes.compute(1000000000);
21     }
22 }
```


Primes / Recursive

```
1  import java.util.*;
2
3  public class Eratosthenes3 {
4      boolean[] is_prime;
5
6      public void compute(int max) {
7          is_prime = new boolean[max+1];
8          for (int i = 0; i < max+1; ++i)
9              is_prime[i] = true;
10         do_compute(max);
11     }
12     protected void do_compute(int max) {
13         if (max <= 2)
14             return;
15         int max_factor = (int) Math.floor(Math.sqrt(max));
16         do_compute(max_factor);
17         for (int i = 2; i <= max_factor; ++i) {
18             if (is_prime[i]) {
19                 for (int j = 2*i; j <= max; j += i) {
20                     is_prime[j] = false;
21                 }
22             }
23         }
24     }
25     public static void main(String[] args) {
26         Eratosthenes3 primes = new Eratosthenes3();
27         primes.compute(1000000000);
28     }
29 }
```

Primes / Good Parallelization

```
1  import java.util.*;
2
3  public class Eratosthenes4 {
4      boolean[] is_prime;
5
6      public void compute(int max) {
7          is_prime = new boolean[max+1];
8          for (int i = 0; i < max+1; ++i)
9              is_prime[i] = true;
10         do_compute(max);
11     }
12     protected void do_compute(int max) {
13         if (max <= 2)
14             return;
15         int max_factor = (int) Math.floor(Math.sqrt(max));
16         do_compute(max_factor);
17         // omp parallel for
18         for (int i = 2; i <= max_factor; ++i) {
19             if (is_prime[i]) {
20                 for (int j = 2*i; j <= max; j += i) {
21                     is_prime[j] = false;
22                 }
23             }
24         }
25     }
26     public static void main(String[] args) {
27         Eratosthenes4 primes = new Eratosthenes4();
28         primes.compute(1000000000);
29     }
30 }
```

implementation	runtime [s]
sequential	10.4
badly parallel	>120.0
recursive	10.0
OK parallel (using 8 cores)	6.7

Matrix Multiplication / Sequential

```
1  public class Matrix {
2      int N, M;
3      double[][] values;
4      public Matrix(int N, int M) {
5          this.N = N; this.M = M;
6          values = new double[N][M];
7      }
8      public void fill_random() {
9          for (int n = 0; n < N; ++n) {
10             for (int m = 0; m < M; ++m) {
11                 values[n][m] = 2 * (Math.random() - 0.5);
12             }
13         }
14     }
15
16     public Matrix mul(Matrix B) throws IllegalArgumentException {
17         if (M != B.N)
18             throw new IllegalArgumentException("Number of columns and rows does not match in mul.");
19         Matrix C = new Matrix(N, B.M);
20         for (int n = 0; n < N; ++n) {
21             for (int m = 0; m < B.M; ++m) {
22                 double val = 0;
23                 for (int k = 0; k < M; ++k) {
24                     val += values[n][k] * B.values[k][m];
25                 }
26                 C.values[n][m] = val;
27             }
28         }
29         return C;
30     }
```

```
32     public static void main(String[] args) {
33         Matrix A = new Matrix(1000, 2000),
34             B = new Matrix(2000, 3000);
35         A.fill_random();
36         B.fill_random();
37         Matrix C = A.mul(B);
38     }
39 }
```

Matrix Multiplication / Parallelization

```
1  public class Matrix {
2      int N, M;
3      double[][] values;
4      public Matrix(int N, int M) {
5          this.N = N; this.M = M;
6          values = new double[N][M];
7      }
8      public void fill_random() {
9          for (int n = 0; n < N; ++n) {
10             for (int m = 0; m < M; ++m) {
11                 values[n][m] = 2 * (Math.random() - 0.5);
12             }
13         }
14     }
15
16     public Matrix mul(Matrix B) throws IllegalArgumentException {
17         if (M != B.N)
18             throw new IllegalArgumentException("Number␣of␣columns␣and␣rows␣does␣not␣match␣in␣mul.");
19         Matrix C = new Matrix(N, B.M);
20         // omp parallel for
21         for (int n = 0; n < N; ++n) {
22             for (int m = 0; m < B.M; ++m) {
23                 double val = 0;
24                 for (int k = 0; k < M; ++k) {
25                     val += values[n][k] * B.values[k][m];
26                 }
27                 C.values[n][m] = val;
28             }
29         }
30         return C;
31     }
32
33     public static void main(String[] args) {
34         Matrix A = new Matrix(1000, 2000),
35             B = new Matrix(2000, 3000);
36         A.fill_random();
37         B.fill_random();
38         Matrix C = A.mul(B);
39     }
40 }
```


Matrix Multiplication / Tiled Sequential

```
16     public Matrix mul(Matrix B) throws IllegalArgumentException {
17         if (M != B.N)
18             throw new IllegalArgumentException("Number␣of␣columns␣and␣rows␣does␣not␣match␣in␣mul.");
19         Matrix C = new Matrix(N, B.M);
20         int T = (int) Math.ceil(Math.sqrt(M));
21         for (int n0 = 0; n0 < N; n0+= T) {
22             for (int m0 = 0; m0 < B.M; m0+= T) {
23                 for (int k0 = 0; k0 < M; k0+= T) {
24                     for (int n = n0; n < Math.min(N, n0+T); ++n) {
25                         for (int m = m0; m < Math.min(M, m0+T); ++m) {
26                             double val = 0;
27                             for (int k = k0; k < Math.min(M, k0+T); ++k) {
28                                 val += values[n][k] * B.values[k][m];
29                             }
30                             C.values[n][m] += val;
31                         }
32                     }
33                 }
34             }
35         }
36         return C;
37     }
```

implementation	runtime [s]
sequential	21.2
parallel	4.5
tiled sequential	3.9
tiled parallel (using 8 cores)	1.2