Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

Java - Concurrent Programming

- Multithreading
- Memory Model
- Synchronization
- Monitors
- Locks

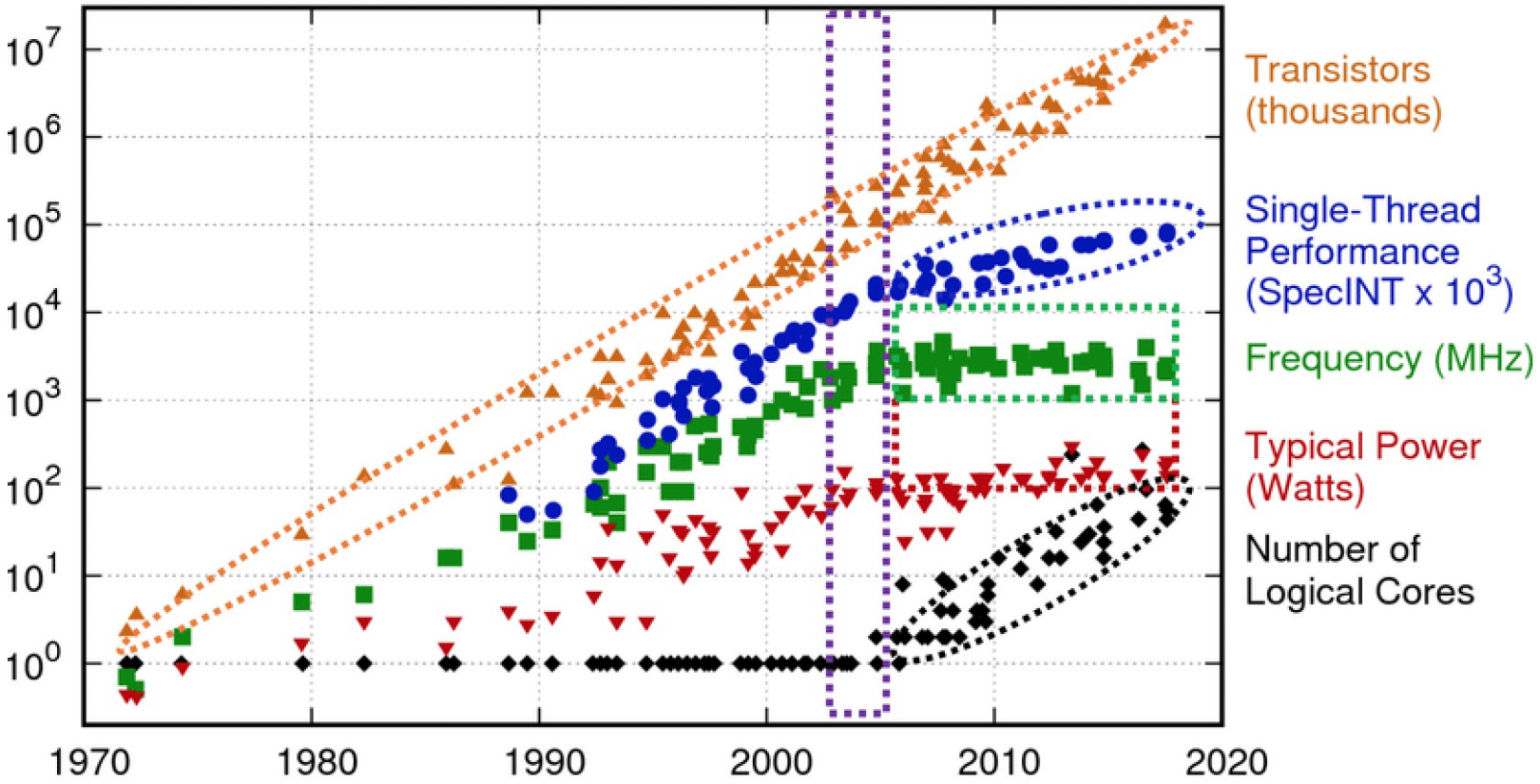
Appendix

- Atomic Access
- Volatile Variables

Literature

- **The Java Tutorials – Lesson: Concurrency**
  https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- Doug Lea. **Concurrent Programming in Java**, Addison-Wesley, 2000.
- The **Java Language Specification**. Java SE 8 Edition. Chapter 17. Threads and Locks. https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html
- Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. **The Java memory model**. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05). http://rsim.cs.uiuc.edu/Pubs/popl05.pdf
- E. Lee, **The Problem with Threads**, Computer, vol. 39, no. 05,2006. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

Evolution of Processors



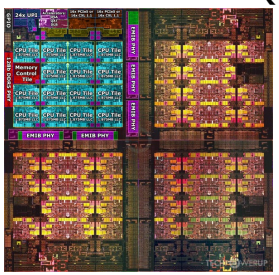
M. Rusanovsky, et al (2019). BACKUS: Comprehensive High-Performance Research Software Engineering Approach for Simulations in Supercomputing Systems.

- Performance increase now mainly through parallelism.
- Sequential programs will not run faster on multicore processors!
- Only parallel programs will run faster on multicore processors.

Everything Parallel

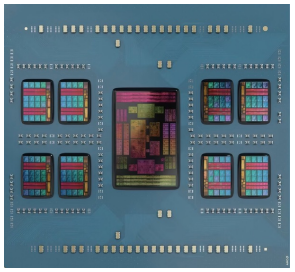
CPUs

Intel Xeon 8490H (2023)



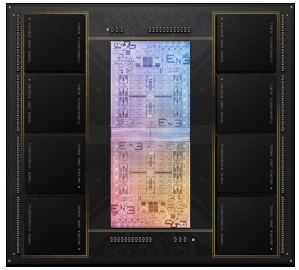
60 cores

AMD EPYC 9754 (2023)



128 cores

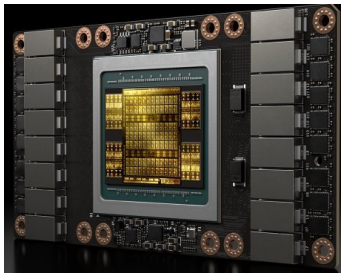
Apple M2 Ultra (2023)



16+8 cores + 76 GPU cores + NE

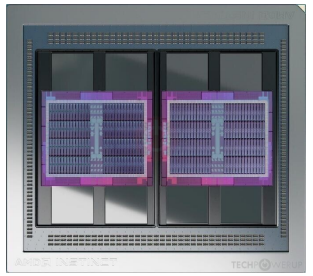
GPUs

NVIDIA Ampere GH100 (2023)



18432 CUDA cores + 576 Tensor cores

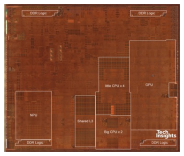
AMD Instinct MI250X (2023)



14080 stream processors

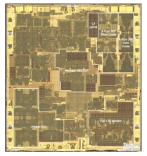
Mobile

Apple A17 (2023)



2+4 cores + 6 GPU cores + NE (16 cores)

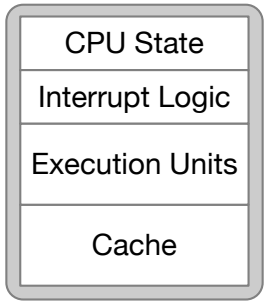
Snapdragon 8 Gen2 (2023)



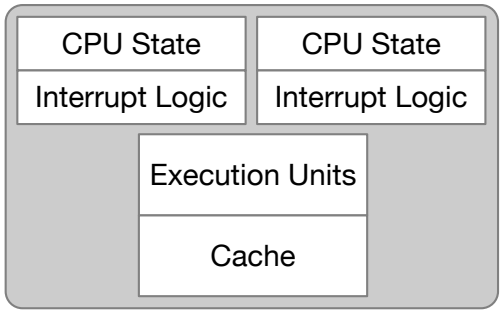
1+2+2+3 cores + GPU + NPU

Processors

- **Processors** (sequential)

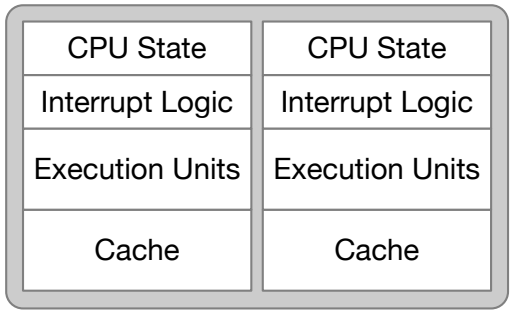


Conventional Processor

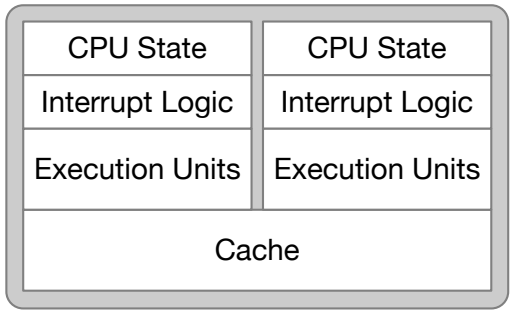


SMT/Hyperthreading Processor

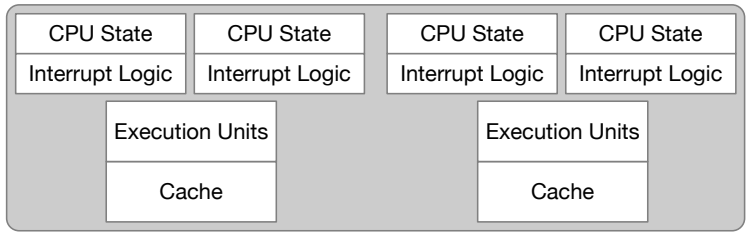
- **Multicore Processors** (parallel)



Multicore Processor
(2 cores, separate caches)

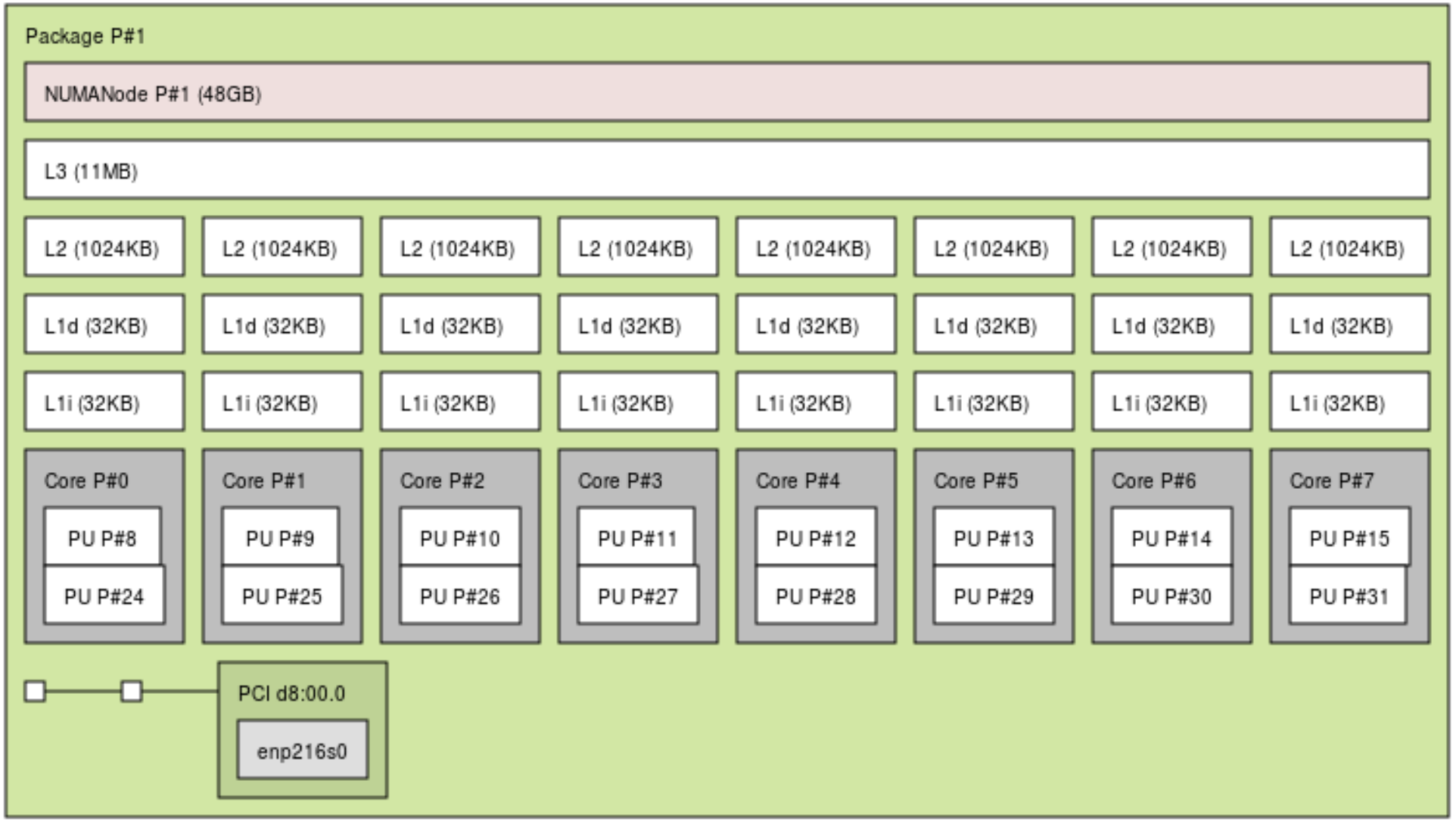


Multicore Processor
(2 cores, shared cache)



Multicore Processor
(2 cores/4 threads, SMT/HT)

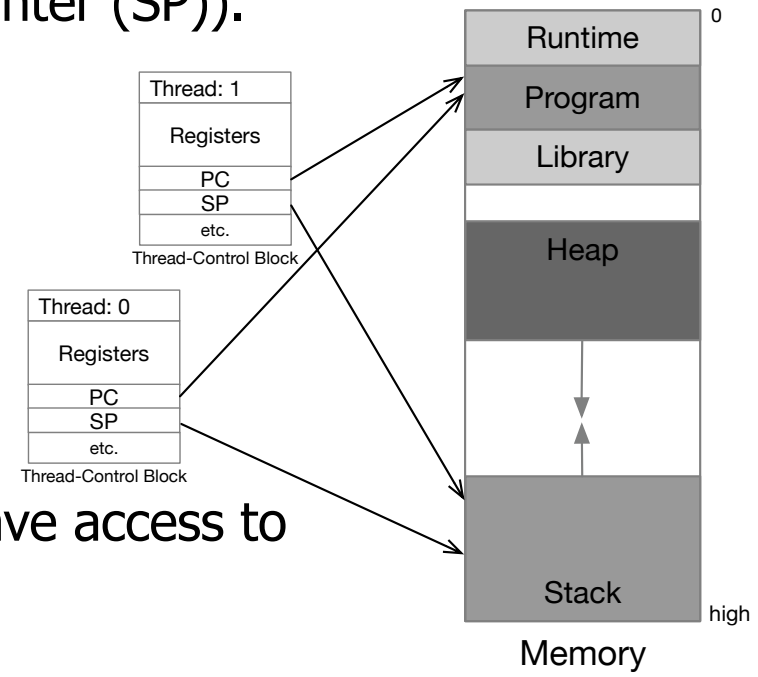
Cache Organization – Intel Xeon 4108



8 Cores / 16 Threads

Multithreading

- A thread is a schedulable sequence of instructions defined by a thread-control block (registers, program counter (PC), stack pointer (SP)).
- A thread can be
  - suspended (save registers in memory)
  - or resumed (restore registers from memory)



- Each thread has its own stack, but all threads have access to shared memory (heap).
- Multiple threads can execute independently
  - in parallel on multiple CPUs/cores (physical concurrency)
  - or arbitrarily interleaved on a single CPU/core (logical concurrency)

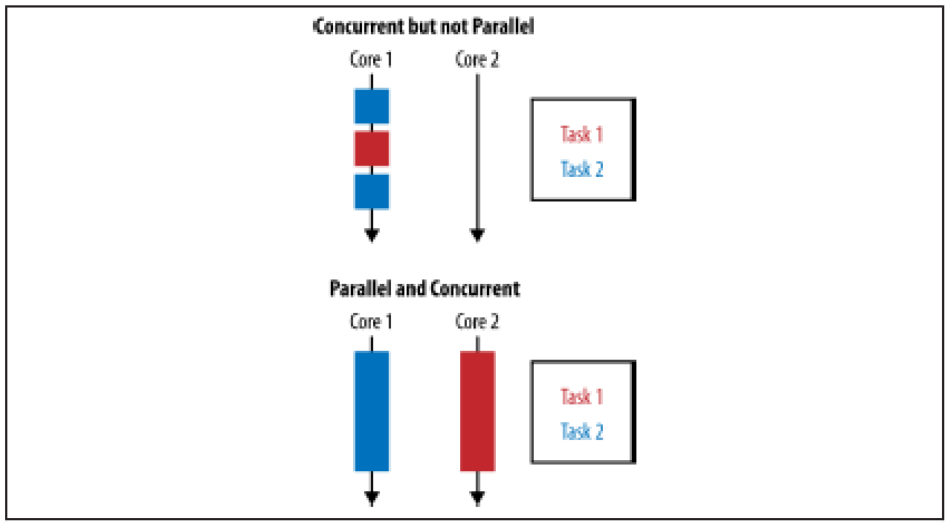
Concurrency vs. Parallelism

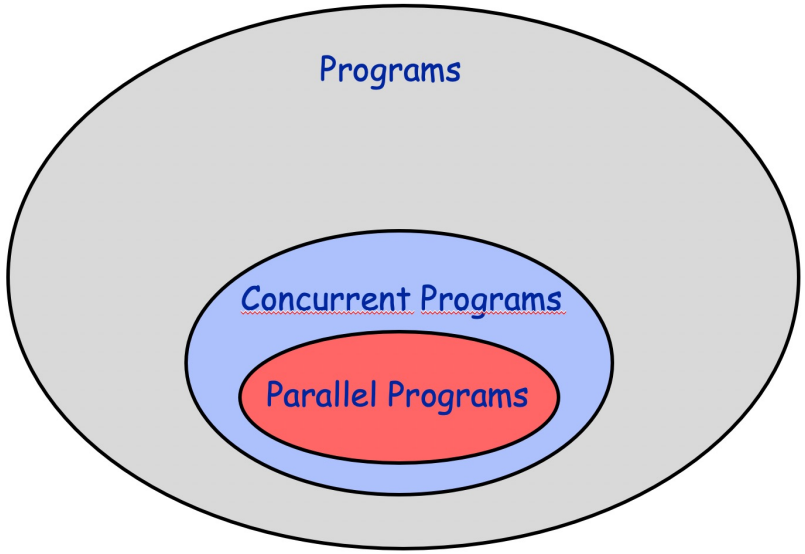
- **Concurrency**

  A condition of a system in which multiple tasks are *logically active* at one time.

- **Parallelism**

  A condition of a system in which multiple tasks are *physically active* at one time.





Java Multithreading

- The Java Virtual Machine can support many threads of execution at once.
- These threads independently execute sequential code that operates on values and objects residing in a shared main memory.
- Threads may be supported by having many cores, by time-slicing a single core, or by time-slicing many cores.

- A class whose instances are intended to be executed by a thread must either
  - extend the class `Thread`
  - or implement the interface `Runnable`.
  - In both cases, such a class must implement the method `run()`.

Generating Threads - Class `Thread`

1. Define a subclass of **`Thread`**
2. In this subclass, override **`run()`**
3. Instantiate the subclass
4. Invoke **`start()`**

```
class MyThread extends Thread {                                    // 1
      public void run() { System.out.println("MyThread"); } // 2
}
public class ThreadTest {
      public static void main(String[] args) {
            MyThread myThread = new MyThread();                    // 3
            myThread.start();                                      // 4
      }
}
```

Generating Threads - Interface `Runnable`

1. Define a new class that implements the interface **`Runnable`**
2. Instantiate this new class.
3. Create an instance of **`Thread`** and pass a reference to the new class that implements the interface **`Runnable`**.
4. Invoke start() on the instance of **`Thread`**

```
class MyRunner implements Runnable {                          // 1
      public void run() {System.out.println("MyRunner"); }
}
public class RunnableTest {
      public static void main(String[] args) {
            MyRunner myRunner = new MyRunner();               // 2
            new Thread(myRunner).start();                     // 3, 4
      }
}
```

Asynchronous Execution

```
public static void main(String[] args) {
    Runnable r = () -> { System.out.println("Some thread"); };
    Thread t1 = new Thread(r);
    t1.start();

    Thread t2 = new Thread(() -> {
        System.out.println("Some other thread");});
    t2.start();

    new Thread(() -> {
        System.out.println("Yet another thread");}).start();
    System.out.println("done.");
    ...
```

Possible output:

```
Some other thread
Some thread
done.
Yet another thread
```

Class Thread – Some Methods

- **`join()`**
  Waits for this thread to die.
- **`start()`**
  Causes this thread to begin execution; the JVM calls the run method.
- **`yield()`**
  A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- **`sleep(long millis)`**
  Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **`notifyAll()`** (inherited from class Object)
  Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.
- **`wait()`** (inherited from class Object)
  Causes the current thread to wait until another thread invokes the **`notify()`** method or the **`notifyAll()`** method for this object.

Shared Variables

- Memory that can be shared between threads is called shared memory or heap memory.
- All instance fields, static fields, and array elements are stored in heap memory.
- Conflicting accesses to shared variables by multiple treads need to be synchronized.
- Local variables, formal method parameters, and exception handler parameters are never shared between threads (allocated on thread stack).

Example: Counter for 2 Threads

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start();
      t2.start();


      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) increment();
   }

   private static void increment() { cnt++; }
}
```

Possible Output: ????

Example: Counter for 2 Threads

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start();
      t2.start();
      t1.join();                  // wait until t1 has finished
      t2.join();                  // wait until t2 has finished
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) increment();
   }

   private static void increment() { cnt++; }
}
```

Possible Output: ????

Example: Counter for 2 Threads

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++){
                increment();
      }
   }

   private synchronized static void increment() { cnt++; }
}
```

```
Output: 2000
```

The Java Memory Model

- The behavior of threads, particularly when not correctly synchronized, can be confusing and counterintuitive.

- The Java Memory Model describes the semantics of multithreaded programs; it includes rules for which values may be seen by a read of shared memory that is updated by multiple threads.

The Java Memory Model

- This program uses local variables `r1` and `r2` and shared variables A and B.
- Initially, `A == B == 0`.

Thread 1	Thread 2
`1: r2 = A;`	`3: r1 = B;`
`2: B = 1;`	`4: A = 2;`

Q: Is the result `r2 == 2` and `r1 == 1` possible?

https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4

The Java Memory Model

- All possible interleavings between Thread 1 and Thread 2?

```
1: r2 = A;
2: B = 1;
3: r1 = B;
4: A = 2;
```

r2 == 0, r1 == 1

```
3: r1 = B;
1: r2 = A;
2: B = 1;
4: A = 2;
```

r2 == 0, r1 == 0

```
3: r1 = B;
1: r2 = A;
4: A = 2;
2: B = 1;
```

r2 == 0, r1 == 0

```
3: r1 = B;
4: A = 2;
1: r2 = A;
2: B = 1;
```

r2 == 2, r1 == 0

```
1: r2 = A;
3: r1 = B;
2: B = 1;
4: A = 2;
```

r2 == 0, r1 == 0

```
1: r2 = A;
3: r1 = B;
4: A = 2;
2: B = 1;
```

r2 == 0, r1 == 0

The Java Memory Model

- This program uses local variables r1 and r2 and shared variables A and B.
- Initially, A == B == 0.

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

Q: Is the result `r2 == 2` and `r1 == 1` possible?

A: YES – because this is an incorrectly synchronized program!

The semantics of the Java programming language allow compilers and microprocessors to perform optimizations that can interact with incorrectly synchronized code in ways that can produce behaviors that seem paradoxical.

See: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

```
2: B = 1;
4: A = 2;
1: r2 = A;
3: r1 = B;
```

r2 == 2, r1 == 1

The Java Memory Model

- The Java Memory Model describes the semantics of multithreaded programs; it includes rules for which values may be seen by a read of shared memory that is updated by multiple threads.

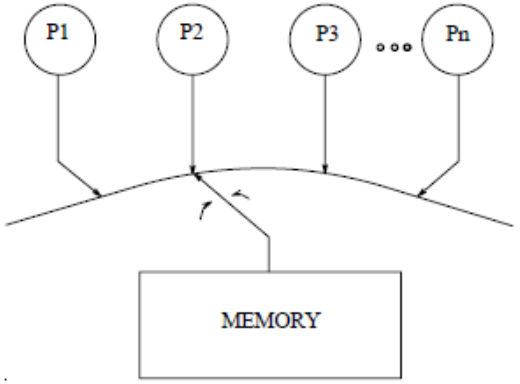
- The new Java memory model (since Java 5) provides a simple interface for correctly synchronized programs:

The Java memory model guarantees sequential consistency to data-race-free programs.

[Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05).]
http://rsim.cs.uiuc.edu/Pubs/popl05.pdf

The Java Memory Model

The Java memory model adopts the data-race-free approach for correct programs – correct programs are those that are data-race-free; such programs are guaranteed sequential consistency.



Sequential consistency

The result of any execution is the same as if each processor's accesses were kept in sequential order (=program order), and accesses among different processors were interleaved. [Lamport, 1979]

The Java Memory Model

Shared Variables

- Memory shared between threads is called shared memory or heap memory.
- All instance fields, static fields, and array elements are stored in heap memory.
- Local variables, formal method parameters, and exception handler parameters are never shared between threads.

Conflicting Access

- Two accesses to (reads of or writes to) the same shared variable are said to be conflicting if at least one of the accesses is a write.

Data Race

- Two accesses to the same shared variable form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by happens-before.

The Java Memory Model

- The Java Memory Model defines the **happens-before** order on memory operations such as reads and writes of shared variables.

- The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation happens-before the read operation.

- The happens-before order is given by the transitive closure of synchronizes-with order (total order over all synchronization actions) and program order.

For more details see: Chapter 17 of the Java Language Specification

Data-Race-Free Program

A program is said to be correctly synchronized or data-race-free if and only if all sequentially consistent executions of the program are free of data races.

- Number of sequentially consistent executions (interleavings) between t threads, each executing s statements*: $scx = \frac{(st)!}{s!^t}$
  - *s=2, t=2; scx= 6*
  - *s=3, t=2; scx = 20*
  - *s=5, t=2; scx= 252*
  - *s=10, t=2; scx= 184.756*
  - *s=15, t=2; scx= 155.117.720*

*assembly language statements or machine instructions

Synchronization - Monitors

- Java provides multiple mechanisms for communicating between threads. The most basic one is synchronization, which is implemented using monitors.

- Each object is associated with a monitor, which a thread can lock or unlock.

- Only one thread at a time may hold a lock on a monitor (intrinsic lock). Any other threads attempting to lock that monitor are blocked until they can obtain the intrinsic lock.

Mutual Exclusion and Visibility

Synchronization has two important aspects:

- Mutual Exclusion

  Only one thread can execute a block of code (critical section) at a time.
- Visibility

  Changes made by one thread to shared data are visible to other threads.

Intrinsic locks play a role in both aspects of synchronization:

- enforcing exclusive access to an object's state and
- establishing happens-before relationships that are essential to visibility.

Synchronization Mechanisms

- **Synchronized statements**

```
...
synchronized (someObjectref) {
    ...
}
```

- **Synchronized methods**

```
class MySynchronizedClass {
  synchronized void someMethod() {
    ...
  }
}
```

- Alternative synchronization mechanisms include, semaphores, atomic variables, volatile variables, barriers, phasers, etc.

Synchronized Statement

- The synchronized statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor and blocks (i.e., does not proceed further) until the lock action has successfully completed.
- After the lock action has been performed, the body of the synchronized statement is executed.
- If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

```
...
synchronized (someObjectref) {
    ...
}
```

Critical Section
Only one thread at a time can enter.

Synchronized Methods

- A synchronized method automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed.

- If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as `this` during execution of the body of the method).
- If the method is static, it locks the monitor associated with the `Class` object that represents the class in which the method is defined.

- If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

Synchronized Methods

- Any two invocations of synchronized methods on the same object never interleave.

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

- When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

See: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Example – Missing Synchronization

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start();
      t2.start();
      t1.join();
      t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) increment();
   }

   private static void increment() { cnt++; }
}
```

Possible Output: 1365

Example – Synchronized Block

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) {
           synchronized (getClass()) {
                increment();
           }
      }
   }

   private static void increment() { cnt++; }
}
```

`getClass()` (from `Object`) returns the runtime class of this Object, which exists exactly once per class, and provides the intrinsic lock.

Critical Section

Only one thread at a time can enter.

```
Output: 2000
```

Example – Synchronized Member Method

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) {
                increment();
      }
   }

   private synchronized void increment() { cnt++; } // WRONG
}
```

Since `increment()` is a member method, each thread acquires the intrinsic lock associated with `this`, i.e., t1 and t2 lock different objects.

Possible Output: 1377

Example – Synchronized Static Method

```
public class ThreadSync extends Thread {
   static int cnt = 0;
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++){
                increment();
      }
   }

   private synchronized static void increment() { cnt++; }
}
```

Since `increment()` now is a static method, the thread acquires the intrinsic lock for the `Class` object associated with the class.

```
Output: 2000
```

Example – Counter (Missing Synchronization)

```
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}
public class CounterTest {
    public static void main(String[] args) throws
                                          InterruptedException {
        Counter counter = new Counter();
        Runnable code = () -> {
           for (int i=0; i < 1000; i++) { counter.increment();}};
        Thread t1 = new Thread(code);
        Thread t2 = new Thread(code);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.value());
    }
}
```

```
Possible Output: 1965
```

Example – Synchronized Counter

```
class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
public class CounterTest {
    public static void main(String[] args) throws
                                          InterruptedException {
        SynchronizedCounter counter = new SynchronizedCounter();
        Runnable code = () -> {
           for (int i=0; i < 1000; i++) { counter.increment();}};
        Thread t1 = new Thread(code);
        Thread t2 = new Thread(code);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.value());
    }
}
```

```
Output: 2000
```

Thread Coordination

- Multiple threads often need to coordinate when accessing shared resources.
- Such coordination is often implemented using guarded blocks within synchronized methods/statements.

- Example: Producer thread puts item into a place (slot); consumer thread takes item from that place (slot).

See: https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Thread Coordination

- A producer thread enters the synchronized method and, if the condition to proceed is not meet, **`wait()`** suspends the thread, which releases the lock and enters the wait set.
- Later, a consumer thread acquires the same lock and invokes **`notifyAll()`**, waking up all threads waiting on that lock.
- Some time after the consumer thread has released the lock, the producer reacquires the lock and resumes by returning from **`wait()`**.

```
synchronized void put(int nr) {
   while (!empty) {     //guard
      try { wait(); }
      catch(...) {}
   }
   slot = nr;
   empty = false;
   notifyAll();
}
```

```
synchronized int get() {
   while (empty) {   // guard
      try { wait(); }
      catch(...){}
   }
   int result = slot;
   empty = true;
   notifyAll();
   return result;
}
```

Example: Place with single slot

```
public class Place {
    private int slot;
    private boolean empty = true;

    public synchronized void put(int nr) {
        while (!empty) {
            try { wait(); }
            catch(InterruptedException e) {}
        }
        slot = nr;
        System.out.println("T" + Thread.currentThread().getId()
            + ": put " + slot);
        empty = false;
        notifyAll();
    }
...
```

Example: Place with single slot (ctd.)

```
...
    public synchronized int get() {
        while (empty) {
            try { wait(); }
            catch(InterruptedException e) {}
        }
        int result = slot;
        System.out.println("T" + Thread.currentThread().getId()
            + ": get " + slot);
        empty = true;
        notifyAll();
        return result;
    }
}
```

Example: Place with single slot (ctd.)

Test program with one producer thread and one consumer thread.

```
public class PlaceTest {
    private static final int N = 1000;

    public static void main(String...args) {
        Place place = new Place();

        Runnable produce = () -> {
            for (int i = 1; i <= N; i++) {
                place.put(i);}};
        Runnable consume = () -> {
            for (int i = 1; i <= N; i++) {
                place.get();}};
        Thread producer = new Thread(produce);
        Thread consumer = new Thread(consume);
        producer.start();
        consumer.start();
    }
}
```

Output:

```
T13: put 1
T14: get 1
T13: put 2
T14: get 2
T13: put 3
T14: get 3
T13: put 4
T14: get 4
T13: put 5
T14: get 5
T13: put 6
T14: get 6
...
T13: put 1000
T14: get 1000
```

Wait/Notify

Wait

- Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` for this object.
- The thread releases ownership of this object's monitor and waits until another thread notifies threads waiting on this object's monitored.
- When the thread can re-obtain ownership of the monitor it resumes execution.

Notify

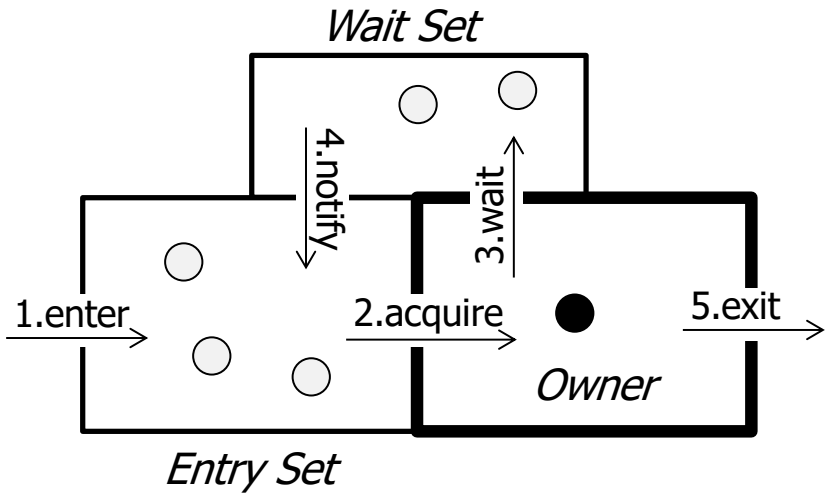
- Wakes up one/all threads waiting on this object's monitor.
- The awakened thread(s) will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread(s) enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

Wait/Notify

- The choice of which thread will obtain ownership of the monitor after **`notify()`** or **`notifyAll()`** has been invoked is arbitrary and occurs at the discretion of the implementation.

→ No fairness guarantees!

Java Monitors – Entry and Wait Set



1. When a thread invokes a synchronized method it enters the entry set competing with other threads (if any) for the lock.
2. One thread from the entry set acquires the lock (becomes the owner) and enters the synchronized method.
3. If the owner calls **`wait()`**, it releases the lock, enters the wait set, and suspends execution.
4. When **`notify/notifyAll()`** is called, one/all thread/s in the wait set compete\s with the threads in the entry set, if any, for the lock.
5. Upon exit from a synchronized method/block the lock is released.

Lock Objects

- Synchronized methods and statements rely on a simple kind of reentrant lock. More sophisticated locking idioms are supported by the package `java.util.concurrent.locks`.

- The most basic interface in this package is the interface `Lock`.

- Lock objects work very much like the implicit locks used by synchronized methods and statements.
  - Only one thread can own a Lock object at a time.
  - Lock objects support a wait/notify mechanism, through their associated `Condition` objects.

Condition Variables

- Conditions (condition queues, condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some condition may now be true.
- A **`Condition`** instance is intrinsically bound to a lock.
- Waiting for a condition atomically releases the associated lock and suspends the current thread, just like **`Object.wait()`**.
- Each condition variable is associated with its own wait set, allowing a more fine-grained coordination and synchronization.
- For each logical condition, there is usually a separate condition variable (e.g. **`bufferEmpty`** and **`bufferNotEmpty`**).

Condition Variables

- To obtain a **`Condition`** instance for a particular **`Lock`** instance use its **`newCondition()`** method.
- The **`await()`** method (or one of its variants) is invoked to wait for a condition.
- The **`signalAll()`** method wakes up all waiting threads. Each thread must re-acquire the lock before it can return from await.

```
Lock lock = new ReentrantLock();
Condition placeEmpty = lock.newCondition();
...
public void put(int nr) throws InterruptedException {
    lock.lock();
    try {
        while (!empty) { placeEmpty.await(); }
        ...
        placeFull.signalAll();
    }
    finally { lock.unlock(); }
}
```

Fairness

- While fairness cannot be achieved with intrinsic locks and wait()/notify() as provided by the class Object, certain types of locks support fairness.

- For example, the constructor for **`ReentrantLock`** accepts an optional fairness parameter.

```
Lock lock = new ReentrantLock(true);
```

- When set true, under contention, such locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.

Liveness Problems

- A concurrent application's ability to execute in a timely manner is known as its liveness.

- **Deadlock** describes a situation where two or more threads are blocked forever, waiting for each other.

- **Starvation** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

- **Livelock** describes a situation where threads are unable to make further progress, although they are not blocked. A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result.

Example – Deadlock

```
public class Deadlock {

static class Friend {
  private final String name;
  public Friend(String name) { this.name = name; }
  public String getName() { return this.name; }

  public synchronized void bow(Friend bower) {
    System.out.format("%s: %s" + "  has bowed to me!%n",
      this.name, bower.getName());
    bower.bowBack(this);
  }
  public synchronized void bowBack(Friend bower) {
    System.out.format("%s: %s" + " has bowed back to me!%n",
      this.name, bower.getName());
  }
}
....
```

Example – Deadlock

```
...
public static void main(String[] args) {
  final Friend alphonse = new Friend("Alphonse");
  final Friend gaston = new Friend("Gaston");
  new Thread(() -> { alphonse.bow(gaston); }).start();
  new Thread(() -> { gaston.bow(alphonse); }).start();
}
```

Possible Scenario:

1. Thread 1 invokes **`bow()`** of **`alphonse`** and acquires implicit lock of **`alphonse`**
2. Thread 2 invokes **`bow()`** of **`gaston`** and acquires implicit lock of **`gaston`**
3. Thread 1 invokes **`bowBack()`** of **`gaston`** → blocks since **`gaston`** is locked
4. Thread 2 invokes **`bowBack()`** of **`alphonse`** → blocks since **`alphonse`** is locked

See: https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Multithreading – Concluding Remarks

- Multithreaded (concurrent) programming is usually much more complex and than sequential programming.

- Threads discard the most essential properties of sequential computation: understandability, predictability, and determinism [E. Lee, 2006].

- Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism [E. Lee, 2006].

Higher-Level Concurrent/Parallel Features

- High-Level Synchronization Primitives
- Tasks and Executor Services
- Thread Pools
- Fork/Join Framework
- Concurrent Collections
- Parallel Streams

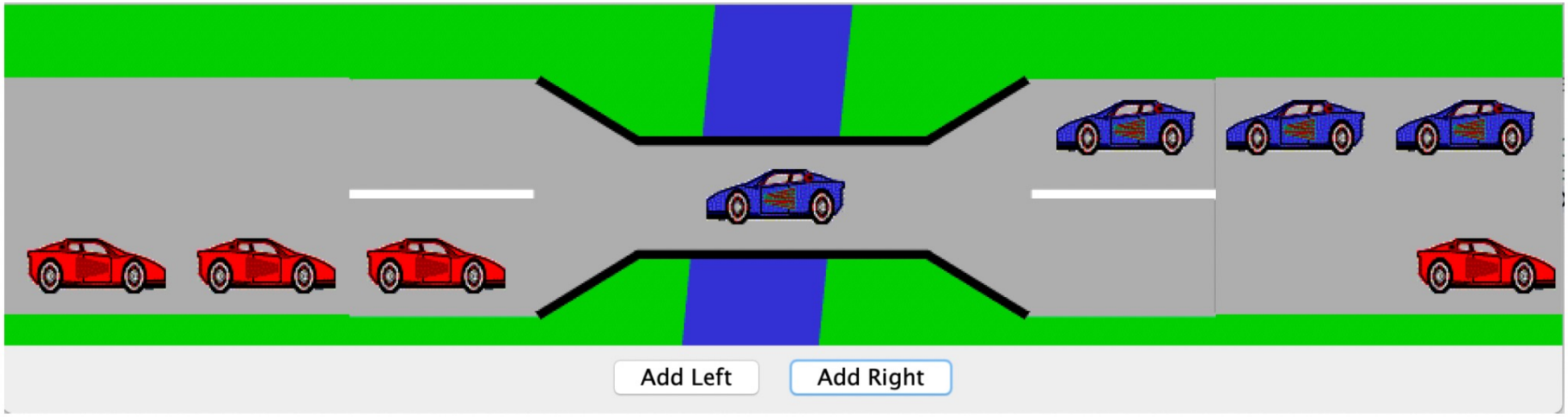
… not covered in this lecture

Assignment 2

Assignment "Traffic Controller"

Develop a traffic controller for controlling the traffic over a single-lane bridge. Two different versions of the traffic controller need to be implemented. The interface **`TrafficController`**, which is given, specifies the functionality of a traffic controller. Moreover, the interface **`TrafficRegistrar`** is provided to register all vehicles that access the bridge, and the interface **`Vehicle`** to specify common properties of all vehicles.

Note: the interfaces *TrafficController, Vehicle und TrafficRegistrar* are given and must not be modified.



Appendix

- **Atomic Access**
- **Volatile Variables**

Atomic Access

- An atomic action is one that effectively happens all at once.
  - An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all.
  - No side effects of an atomic action are visible until the action is complete.
- Reads and writes are atomic
  - for reference variables and
  - for most primitive variables (all types except long and double).
  - for all variables declared volatile (including long and double variables).
- There are special classes to support atomic operations on single variables., e.g., `AtomicInteger`.

Example – Atomic Integer

```
public class ThreadSync extends Thread {
   static AtomicInteger cnt = new AtomicInteger(0);
   public static void main(String[] args) throws
                                          InterruptedException {
      Thread t1 = new ThreadSync();
      Thread t2 = new ThreadSync();
      t1.start();
      t2.start();
      t1.join();
      t2.join();
      System.out.println(cnt);
   }
   public void run() {
      for (int i=0; i < 1000; i++) increment();
   }

   private static void increment() { cnt.incrementAndGet(); }
}
```

```
Output: 2000
```

Volatile Variables

- In addition to monitors, Java provides volatile variables for synchronization.
  - The volatile modifier guarantees that any thread that reads a field will see the most recently written value.
  - Volatile variables are never cached but always read from/written to memory.
  - Reads and writes of volatile variables must not be reordered.

Volatile Variables

- In addition to monitors, Java provides volatile variables for synchronization.
  - The volatile modifier guarantees that any thread that reads a field will see the most recently written value.
  - Volatile variables are never cached but always read from/written to memory.
  - Reads and writes of volatile variables must not be reordered.

- For example, in the following (broken) code fragment, assume that **`this.done`** is a non-volatile boolean field:

```
while (!this.done) Thread.sleep(1000);
```

- The compiler is free to read **`this.done`** just once, and reuse the cached value in each execution of the loop. Thus the loop would never terminate, even if another thread changed the value of **`this.done`**.

Volatile Variables

```
volatile boolean shutdownRequested;
...
public void shutdown() { shutdownRequested = true; }

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

- If volatile is not used, the compiler is free to read **`shutdownRequested`** just once, and reuse the cached value in each execution of the loop. Thus the loop would never terminate, even if another thread changed the value of **`shutdownRequested`**.