Software Engineering

Concurrency

Prof. Agostino Poggi

Problem

- **Synchronize** the **activities** of an application that naturally should be **performed** in **parallel**
  - **Draw** and **display** images on **screen**
  - Check **keyboard** and **mouse input**
  - **Send** and **receive data** on network
  - **Read** and **write** files to disk
  - Perform **useful computation**
- **Avoiding conflicts** in the use of **resources**

Solution

- Use of
  - **Multitasking**
  - **Multiprocessing**
  - **Multithreading**
- Use of
  - An appropriate set of **synchronization supports**

Process Vs Thread

Process

- Is an **executable program** loaded in memory
- Has **own address space**
- **Completely executes** a program
- **Communicate** via **files**, **network**, …
- May **contain** multiple **threads**

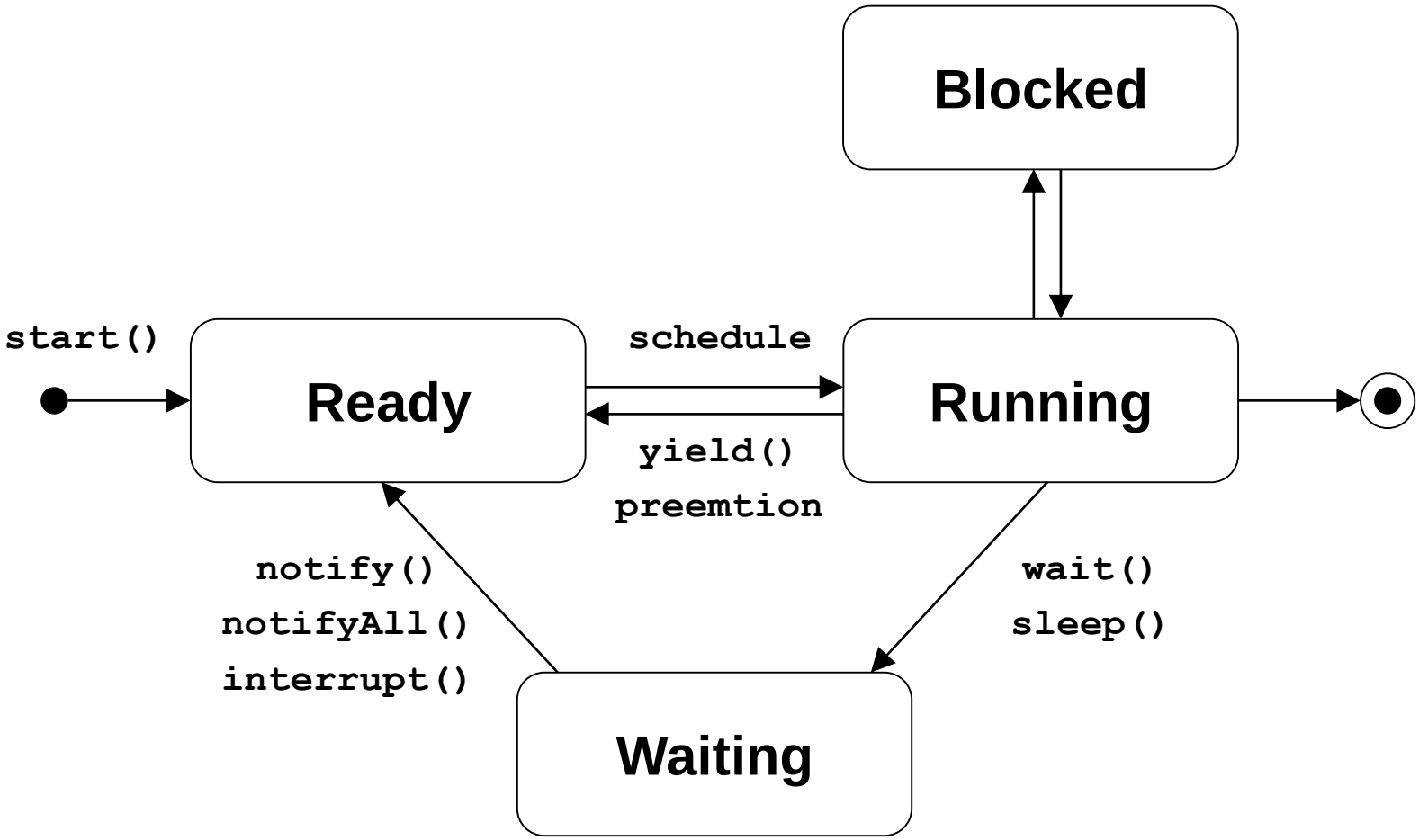
Thread

- **Sequentially** executed **stream** of **instructions**
- **Shares address space** with other threads, but has own execution context
- Each thread **executes** a **part** of the program
- **Communicate** via **shared** access to data
- Also known as "**lightweight process**"

Thread Main Methods

- **start() starts** the thread **execution**
- **run() defines** the thread **task**
- **yield() informs** the scheduler that the current thread is willing to **yield** its current **use** of a **processor**
- **interrupt() interrupts** the current thread
- **setPriority() sets** the **priority** of the current thread
- **sleep() stops** the current thread for a specified **number** of **milliseconds**
- **wait() stops** the current thread **until** another **thread sends it** a **wake up notification** on **its object monitor**
- **notify()** and **notifyAll()** respectively **wake up** one or all the other **threads waiting** on **its object monitor**
- **wait(), notify()** and **notifyAll()** are inherited from the **Object** class

Java Thread Life-Cycle



ThreadExtension Class

```
public final class ThreadExtension extends Thread
{
  private static final int MAXTHREADS = 10;
  private static final int MAXTIME = 1000;

  private static final Random RANDOM  = new Random();

  private ThreadExtension()
  {
  }

  @Override
  public void run()
  {
    try
    {
      Thread.sleep(RANDOM.nextInt(MAXTIME));
    }
    catch (Exception e)
    {
      System.out.println("thread fails!");
    }

    System.out.println("thread en
  }
}
```

```
public static void main(final String[] args)
{
  int n = RANDOM.nextInt(MAXTHREADS);

  System.out.println(n + " threads will be started!");

  for (int i = 0; i < n; i++)
  {
    new ThreadExtension().start();
  }
}
```

ThreadWithRunnable Class

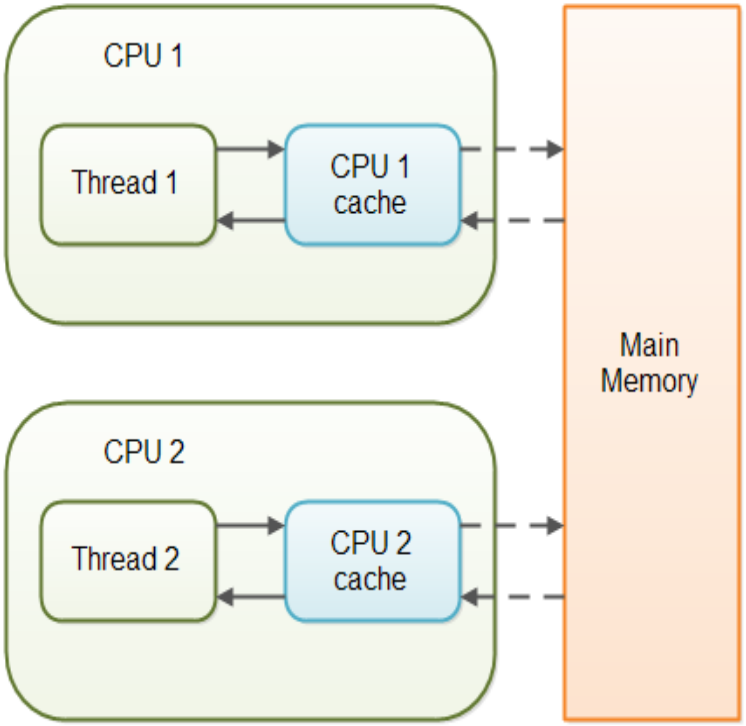
```
public final class ThreadWithRunnable
{
  private static final int MAXTHREADS = 10;
  private static final int MAXTIME = 1000;

  private static final Random RANDOM  = new Random();

  private ThreadWithRunnable()
  {
  }

  public static void main(final St
  {

  }
}
```

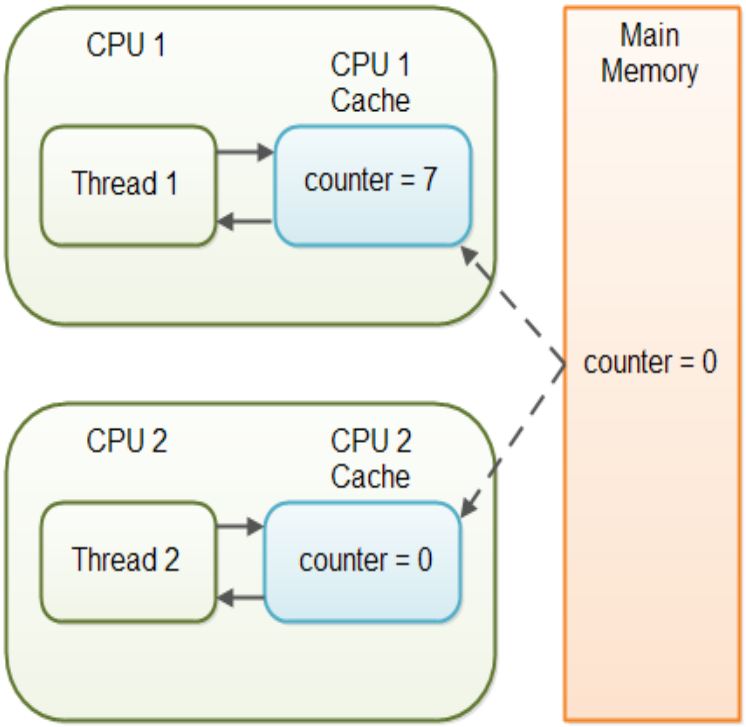
```
class MyRunnable implements Runnable
{
  @Override
  public void run()
  {
    try
    {
      Thread.sleep(RANDOM.nextInt(MAXTIME));
    }
    catch (Exception e)
    {
      System.out.println("thread fails!");
    }

    System.out.println("thread ends!");
  }
}

int n = RANDOM.nextInt(MAXTHREADS);

System.out.println(n + " threads will be started!");

for (int i = 0; i < n; i++)
{
  new Thread(new MyRunnable()).start();
}
```

ThreadWithLambda Class

```
public final class ThreadWithLambda
{
  private static final int MAXTHREADS = 10;
  private static final int MAXTIME = 1000;

  private static final Random RANDOM  = new Random();

  private ThreadWithLambda()
  {
  }

  public static void main(final String[] args)
  {
    int n = RANDOM.nextInt(MAXTHREADS);

    System.out.println(n + " threads will be started!");

    Runnable runnable = () -> {
      try
      {
        Thread.sleep(RANDOM.nextInt(MAXTIME));
      }
      catch (Exception e)
      {
        System.out.println("thread fails!");
      }

      System.out.println("thread ends!");
    };

    for (int i = 0; i < n; i++)
    {
      new Thread(runnable).start();
    }
  }
}
```

Caching Problems





Volatile Variables

- The **volatile keyword** is intended to address **variable visibility** problems
- All the **writes** to a **volatile variable** are **written** back to **main memory** immediately

- All the **reads** of a **volatile variable** are **read** directly from **main memory**
- **Reading** and **writing** of volatile variables causes the variable to be read or written to **main memory** and it is **more expensive** than accessing the CPU cache
- Accessing **volatile variables** also **prevent instruction reordering** which is a normal **performance enhancement** technique

NonVolatileDemo & VolataliDemo Classes

```
public class NonVolatileDemo extends Thread
{
  private boolean keepRunning = true;

  public NonVolatile
  {
    this.keepRunning
  }

  @Override
  public void run()
  {
    System.out.print

    long t = System.

    while (keepRunn
    {
    }

    System.out.print
  }

  public static voi
  {
    NonVolatileDemo

    demo.start();
    Thread.sleep(10

    demo.keepRunning
  }
}
```

```
Thread terminated in
```

```
public class VolatileDemo extends Thread
{
  private volatile boolean keepRunning = true;

  public VolatileDemo()
  {
    this.keepRunning = true;
  }

  @Override
  public void run()
  {
    System.out.print("\n Thread terminated in ");

    long t = System.currentTimeMillis();

    while (keepRunning)
    {
    }

    System.out.println((System.currentTimeMillis() - t) + " milliseconds");
  }

  public static void main(final String[] args) throws InterruptedException
  {
    VolatileDemo demo = new VolatileDemo();

    demo.start();
    Thread.sleep(1000);

    demo.keepRunning = false;
  }
}
```

```
Thread terminated in 1002 milliseconds
```

Atomicity Problems

- The operation **i++** might **seem** like an **atomic operation**, but it is actually not. In fact, it consists of **three operations**
  - **Read** operation, where the value of i is read
  - **Modify** operation, where a new value is being calculated (i + 1)
  - **Write** operation, where the new value is written to the variable i
- This is a problem when **different threads** works on the **same variable** or in general on the **same resources**

ThreadRaceDemo Class

```
public final class ThreadRaceDemo
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;

  private static int shared = 0;

  private ThreadRaceDemo()
  {
  }

  public static void main(final String[] args)
  {
    Runnable runnable = () -> {
      for (int i = 0; i < OPERATIONS; i++)
      {
        shared++;
      }
    };

    for (int j = 0; j < THREADS; j++)
    {
      new Thread(runnable).start();
    }

    System.out.println("\n shared value is: " + shared);
  }
}
```

```
shared value is: 92945
```

Sometimes the execution of operations overlaps!

Some threads might still run when the println method is executed!

SynchronizedRaceDemo Class

```
public class SynchronizedRaceDemo extends Thread
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;

  private static int shared = 0;

  private Thread thread;

  public SynchronizedRaceDemo(final Thread t)
  {
    this.thread = t;
  }

  @Override
  public void run()
  {
    for (int i = 0; i < OPERATIONS; i++)
    {
      shared++;
    }

    joinThread(this.thread);
  }

  private static void joinThread(final Thread t)
  {
  }

  public static void main(final String[] args)
  {
  }
}
```

```
if (t != null)
{
  try
  {
    t.join();
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

```
shared value is: 91030
```

```
Thread t = null;

for (int j = 0; j < THREADS; j++)
{
  t = new SynchronizedRaceDemo(t);

  t.start();
}

joinThread(t);

System.out.println(
  "\n shared value is: " + shared);
```

Creates a chain of threads where each of them waits for the end of the previous one

ThreadPoolRaceDemo Class

```
public final class ThreadPoolRaceDemo
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;
  private static final long SLEEPTIME = 10;


  private static int shared = 0;

  private ThreadPoolRaceDemo()
  {
  }

  public static void main(final String[] arg
  {

  }
}
```

```
Runnable runnable = () -> {
  for (int i = 0; i < OPERATIONS; i++)
  {
    shared++;
  }
};
```

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());
```

```
for (int j = 0; j < THREADS; j++)
{
  pool.execute(runnable);
}
```

```
while (pool.getActiveCount() > 0)
{
  try
  {
    Thread.sleep(SLEEPTIME);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

```
System.out.println("\n shared value is: " + shared);
```

```
shared value is: 63074
```

Synchronized Methods and Blocks

- When a **thread** is *executing* a **synchronized method** for an **object**, **all other threads** that invoke **synchronized methods** on the **same object** suspend their **execution until** the **first thread completes** the **execution** of the synchronized method

- When a **thread** is **executing** the code of a **synchronized block**, all other threads attempting to entering in the **synchronized block** are **blocked** until the **thread** inside the synchronized block **exits** from the **block**

SynchronizedMethodStaticRaceDemo Class

```
public class SynchronizedMethodStaticRaceDemo extends Thread
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;
  private static final long SLEEPTIME = 10;

  private static int shared = 0;

  public synchronized void increment()
  {
    shared++;
  }

  @Override
  public void run()
  {
    for (int i = 0; i < OPERATIONS; i++)
    {
      increment();
    }
  }

  public static void main(final String[]
  {

  }
}
```

```
shared value is: 88240
```

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());

for (int j = 0; j < THREADS; j++)
{
  pool.execute(new SynchronizedMethodStaticRaceDemo());
}

while (pool.getActiveCount() > 0)
{
  try
  {
    Thread.sleep(10);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

System.out.println("\n shared value is: " + shared);
```

SynchronizedMethodRaceDemo Class

```
public class SynchronizedMethodRaceDemo extends Thread
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;

  private static Shared shared = new Shared();

  protected static class Shared
  {
    protected int value;

    protected Shared()
    {
      this.value = 0;
    }

    public synchronized void increment()
    {
      this.value++;
    }
  }

  @Override
  public void run()
  {

  }

  public static void main(final String[] ar
  {

  }
}
```

```
shared value is: 100000
```

```
for (int i = 0; i < OPERATIONS; i++)
{
  shared.increment();
}
```

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());

for (int j = 0; j < THREADS; j++)
{
  pool.execute(new SynchronizedMethodStaticRaceDemo());
}

while (pool.getActiveCount() > 0)
{
  try
  {
    Thread.sleep(10);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

System.out.println("\n shared value is: " + shared.value);
```

SynchronizedBlockRaceDemo Class

```
public class SynchronizedBlockRaceDemo extends Thread
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;
  private static final long SLEEPTIME = 10;
  private static Shared shared = new Shared();

  protected static class Shared
  {
    protected int value;

    protected Shared()
    {
      this.value = 0;
    }

    public void increment()
    {
      // ... some code ...
      synchronized (this)
      {
        this.value++;
      }
      // ... some code ...
    }
  }

  @Override
  public void run()
  {
    for (int i = 0; i < OPERATIONS; i++)
    {
      shared.increment();
    }
  }

  public static void main(final String[] args)
  {
    ThreadPoolExecutor pool = new ThreadPoolExecutor(
        COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());

    for (int j = 0; j < THREADS; j++)
    {
      pool.execute(new SynchronizedMethodRaceDemo());
    }

    while (pool.getActiveCount() > 0)
    {
      try
      {
        Thread.sleep(SLEEPTIME);
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }
}
```

```
shared value is: 100000
```

SynchronizedExternalBlockRaceDemo Class

```
public class SynchronizedExternalBlockRaceDemo extends Thread
{
   private static final int THREADS = 100;
   private static final int OPERATIONS = 1000;
   private static final int COREPOOL = 5;
   private static final int MAXPOOL = 100;
   private static final long IDLETIME = 5000;
   private static final long SLEEPTIME = 10;
   private static Shared shared = new Shared();

   protected static class Shared
   {
      protected int value;

      protected Shared()
      {
         this.value = 0;
      }
   }

   public void increment()
   {
      // ... some code ...
      synchronized (shared)
      {
         shared.value++;
      }
      // ... some code ...
   }
}
```

```
shared value is: 100000
```

```
@Override
public void run()
{
   for (int i = 0; i < OPERATIONS; i++)
   {
      shared.increment();
   }
}
```

```
public static void main(final String[] args)
{
   ThreadPoolExecutor pool = new ThreadPoolExecutor(
         COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
         new LinkedBlockingQueue<Runnable>());

   for (int j = 0; j < THREADS; j++)
   {
      pool.execute(new SynchronizedMethodRaceDemo());
   }

   while (pool.getActiveCount() > 0)
   {
      try
      {
         Thread.sleep(SLEEPTIME);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
```

Atomic Object Classes

- Provides **atomic methods**, i.e., methods that their **instructions** are all **executed together**
- Either all of them **executed together**, or **none** of them **executed**
- Atomic method only **modifies data** of its **own object** without side effects
- Are defined in the **java.util.concurrent.atomic** package

Atomic Object Classes

AtomicBoolean	AtomicReference<V>
AtomicInteger	AtomicReferenceArray<E>
AtomicIntegerArray	AtomicReferenceFieldUpdater<T, V>
AtomicIntegerFieldUpdater<T>	AtomicStampedReference<V>
AtomicLong	DoubleAccumulator
AtomicLongArray	DoubleAdder
AtomicLongFieldUpdater<T>	LongAccumulator
AtomicMarkableReference<V>	LongAdder

Atomic Object Main Methods

- **addAndGet()** adds the given **value** to the **current value**
- **decrementAndGet()** decrements by **one** the **value**
- **incrementAndGet()** increments by **one** the **value**
- **accumulateAndGet()** updates the current value with the **results** of applying the **given function** to the **current** and **given** values
- **lazySet(i) eventually sets** to the **given value**

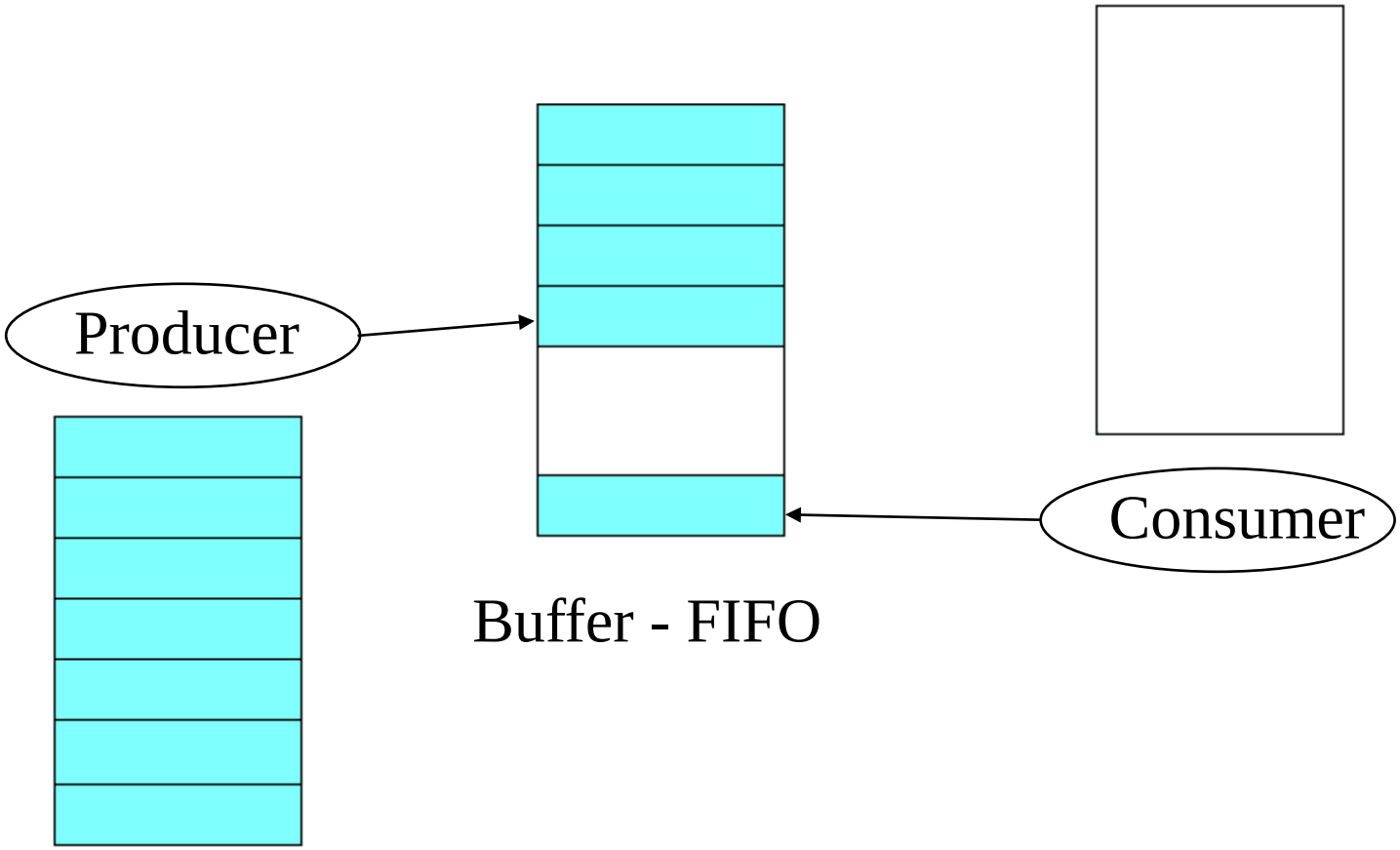
AtomicRaceDemo Class

```
public class AtomicRaceDemo extends Thread
{
  private static final int THREADS = 100;
  private static final int OPERATIONS = 1000;
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;
  private static final long SLEEPTIME = 10;

  private static AtomicInteger shared = new AtomicInteger(0);

  @Override
  public void run()
  {
    for (int i = 0; i < OPERATIONS;
    {
      shared.incrementAndGet();
    }
  }

  public static void main(final Str
  {

  }
}
```

```
shared value is: 100000
```

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());

for (int j = 0; j < THREADS; j++)
{
  pool.execute(new AtomicRaceDemo());
}

while (pool.getActiveCount() > 0)
{
  try
  {
    Thread.sleep(SLEEPTIME);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

System.out.println("\n shared value is: " + shared.get());
```

Producer - Consumer Problem



Producer - Consumer Problem Classifiers

```java
public interface Buffer
{
  int size();

  String get();

  void put(String s);
}
```

```java
public class Producer implements Runnable
{
  private Buffer data;
  private int products;

  public Producer(final Buffer d, final int n)
  {
    this.data     = d;
    this.products = n;
  }

  @Override
  public void run()
  {
    for (int i = 0; i < this.products; i++)
    {
      this.data.put(String.valueOf(i));
    }

    System.out.println(
      "Producer managed " + this.products + " produts");
  }
}
```

```java
public class Consumer implements Runnable
{
  private Buffer data;
  private int products;


  public Consumer(final Buffer d, final int p)
  {
    this.data     = d;
    this.products = p;
  }

  @Override
  public void run()
  {
    for (int i = 0; i < this.products; i++)
    {
      this.data.get();
    }

    System.out.println(
      "Comumer managed " + this.products + " produts");
  }
}
```

SynchronizedMethodBuffer Class

```
public class SynchronizedMethodBuffer implements Buffer
{
  private List<String> elements;

  private final int length;

  public SynchronizedMethodBuffer(final int l)
  {
    this.elements = new ArrayList<>();

    this.length = l;
  }

  @Override
  public int size()
  {
    return this.elements.size();
  }
}
```

```
@Override
public synchronized void put(final String s)
{
  if (this.elements.size() < this.length)
  {
    this.elements.add(s);
  }
  else
  {
    // what operations should be done?
  }
}

@Override
public synchronized String get()
{
  if (this.elements.size() > 0)
  {
    return this.elements.remove(0);
  }
  else
  {
    return null;
  }
}
```

SynchronizedMethodWaitSignalBuffer Class

```
public class SynchronizedMethodWaitSignalBuffer implements Buffer
{
  private List<String> elements;

  private final int length;

  public SynchronizedMethodWaitSignalBuffer(final int l)
  {
    this.elements = new ArrayList<>();

    this.length = l;
  }

  @Override
  public int size()
  {
    return this.elements.size();
  }
}
```

```
@Override
public synchronized String get()
{
  while (this.elements.size() == 0)
  {
    try
    {
      wait();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  notifyAll();

  return this.elements.remove(0);
}

@Override
public synchronized void put(final String s)
{
  while (this.elements.size() == this.length)
  {
    try
    {
      wait();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
  }

  this.elements.add(s);
  notifyAll();
}
```

Concurrent Collection Main Classes (1/2)

- **CopyOnWriteArrayList** is an implementation backed up by a **copy-on-write array**. No synchronization is **necessary**, even during iteration

- **CopyOnWriteArraySet** is an implementation backed up by a **copy-on-write array.** No synchronization is **necessary**, even during iteration

- **ConcurrentHashMap** is a highly concurrent, high-performance implementation backed up by a **hash table**

Concurrent Collection Main Classes (1/2)

- **LinkedBlockingQueue**, **ArrayBlockingQueue** and **PriorityBlockingQueue** are implementations providing methods able to **wait** for **data**

- **DelayQueue** is a blocking queue of delayed elements, in which an **element** can only **be taken** when its **delay** has **expired**

- **SynchronousQueue** is a blocking queue in which each **insert operation** waits for a corresponding **remove operation** by **another thread**, and vice versa

BlockingQueueBuffer Class

```
public class BlockingQueueBuffer implements Buffer
{
  private ArrayBlockingQueue<String> elements;

  public BlockingQueueBuffer(final int l)
  {
    this.elements = new ArrayBlockingQueue<String>(l);
  }

  @Override
  public int size()
  {
    return this.elements.size();
  }
}
```

```
@Override
public String get()
{
  try
  {
    return this.elements.take();
  }
  catch (InterruptedException e)
  {
    return null;
  }
}

@Override
public void put(final String s)
{
  try
  {
    this.elements.put(s);
  }
  catch (InterruptedException e)
  {
    e.printStackTrace();
  }
}
```

Waits for a new item

Waits for a space

BufferDemo Class

```
public final class BufferDemo
{
  private static final int COREPOOL = 5;
  private static final int MAXPOOL = 100;
  private static final long IDLETIME = 5000;
  private static final int PRODUCTS = 1000;
  private static final long SLEEPTIME = 10;
  private static final int BUFFERSIZE = 10;
  private static final int NODES = 10;

  private BufferDemo()
  {
  }

  public static void main(final String[] args)
  {
  }
}
```

```
System.out.println("Enter:");
System.out.println(
    " w for using a buffer with wait and signal syncronization");
System.out.println(
    " b for using a buffer with blocking queue syncronization");

Scanner scanner = new Scanner(System.in);

String s = scanner.next();

scanner.close();

Buffer b;

if (s.equals("w"))
{
  b = new SynchronizedMethodWaitSignalBuffer(BUFFERSIZE);
}
else
{
  b = new BlockingQueueBuffer(BUFFERSIZE);
}
```

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    COREPOOL, MAXPOOL, IDLETIME, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());

for (int j = 0; j < NODES; j++)
{
  pool.execute(new Producer(b, PRODUCTS));
  pool.execute(new Consumer(b, PRODUCTS));
}

while (pool.getActiveCount() > 0)
{
  try
  {
    Thread.sleep(SLEEPTIME);
  }
  catch (InterruptedException e)
  {
    e.printStackTrace();
  }
}

pool.shutdown();
```