# Multithreading

ID2010

2022

# Threads

Threads allow concurrent execution in one program

Threading is a programming language construct

# Concurrent vs Parallel

- Concurrency : hard
  - distributed systems, hosts, processes, **threads**
  - concurrent algorithms and transactions
  - communication
- Limited Parallelism 2-16 cores: easy
  - multiple cores, GPU, n copies of the same problem
- Extreme Parallelism 16+ cores: hard
  - intra-system communication speed

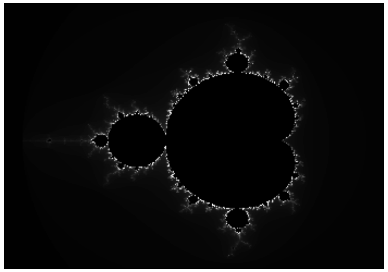
# Parallelism tradeoffs

- Libraries for splitting work units
- Orchestration incurs overhead
- Serial : $t(n) = n$
- Parallel : $t(n) = pre(n,c) + n/c + post(n,c)$

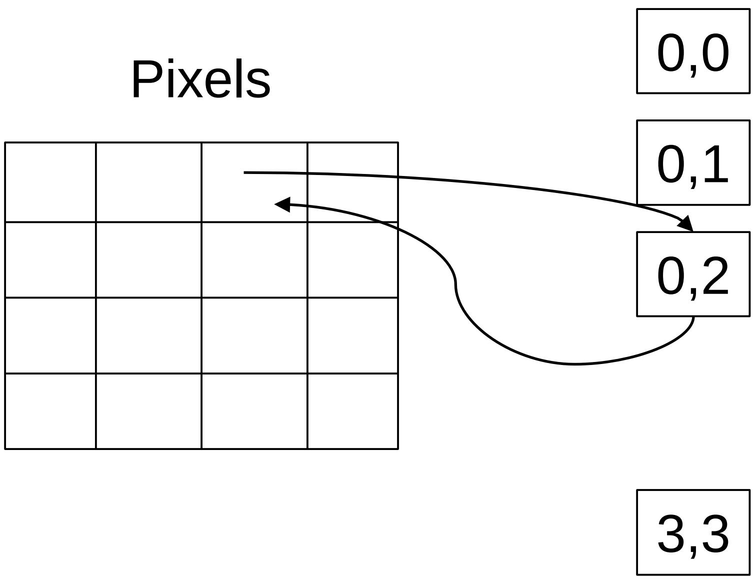
- Parallel may be slower for small data sets

# Example: fractal – single thread

```
void drawMandelBrot (Graphics g, int width, int height) {

    // map each pixel to a complex number and iterate

    for (int y = 0; y < height; y++) {     // vert pixels
        ci = imag(y);                       // imag. part
        for (int x = 0; x < width; x++) { // horz pixels
            cr = real(x);                   // real part
            int iterCount = iterateMandel(cr, ci, limit);
            g.setColor(mapToColor(iterCount));
            g.fillRect(x, y, 1, 1);
        }
    }
}
```



# Example: fractal – single thread




iterCount = Mandel(imag(0), real(0))

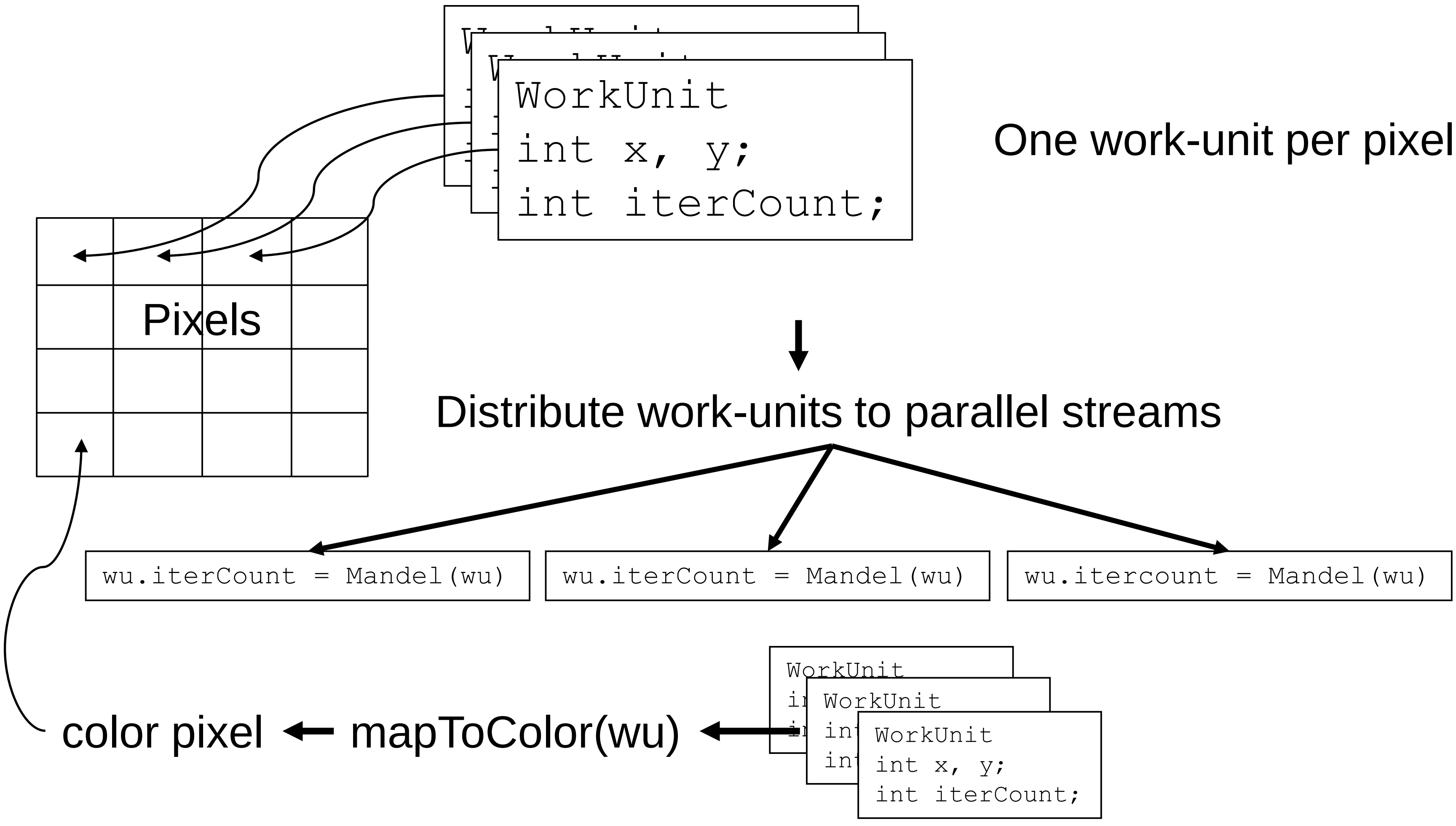
iterCount = Mandel(imag(0), real(1))

iterCount = Mandel(imag(0), real(2))

⋮

itercount = Mandel(imag(3), real(3))

# Example: fractal – par. streams



# Example: fractal – par. streams

```
void drawMandelBrot (Graphics g, int width, int height) {

    configureWorkUnits(width, height); // install cr, ci


    StreamSupport.stream(Arrays.spliterator(workUnits), true)
    .forEach(this::iterateMandel);


    for (WorkUnit wu : workUnits) {
        g.setColor(mapToColor(wu.s));
        g.fillRect(wu.x, wu.y, 1, 1);
    }
}
```

# Threads

Threads allow concurrent execution in one pgm

The main thread is issued by the operating system

It enters the main routine of the program

When it exits the program ends (unless other threads are still running)

# Threads

Threads allow concurrent execution in one pgm

Other threads are issued from the main thread.

They enter **other routines** of the same program.

Several threads can enter **the same routine**.

When the entry point is exited, **the thread ends**.

# Threads

Threads allow concurrent execution in one pgm

The main thread always enters the main program.

```
int foo() {
  a = b;
  c = d;
  ...
}

void bar (int x) {
  x = u * b;
  ...
}

void main (String argv[]) {
  boolean o;
  double p;
  ...
}
```

Other threads execute with other routines as their main programs.

# Threads

Threads allow concurrent execution in one pgm

All threads share global variables.

Threads *do not* share local variables because each thread has its own stack.

```
int a = 0;
boolean b;

void foo() {
  int a;
  ...
}

void bar (int x) {
  float y;
  ...
}

void main (String argv[])
{
  a = 42;
  ...
}
```

Threads can be implemented in several ways:

Virtual threads by interpreter

Parallel processes from OS

Threads supported by OS

# Mental models

- Single-threaded – the whole program
- Multi-threaded – several small programs

- Where and how can and should threads interact in my program?

# Hidden multi-threading:

- Callback APIs
  - GUI events
  - Messaging systems
  - Discovery systems
- Timers
- Parallel streams (Java 8)
- RMI server

# Threads in Java

A class that supports threading implements interface Runnable.

Interface Runnable has one method: run(). This is a thread's entry point.

A new thread is created just like any other object. It is given the object where to execute and told to start running.

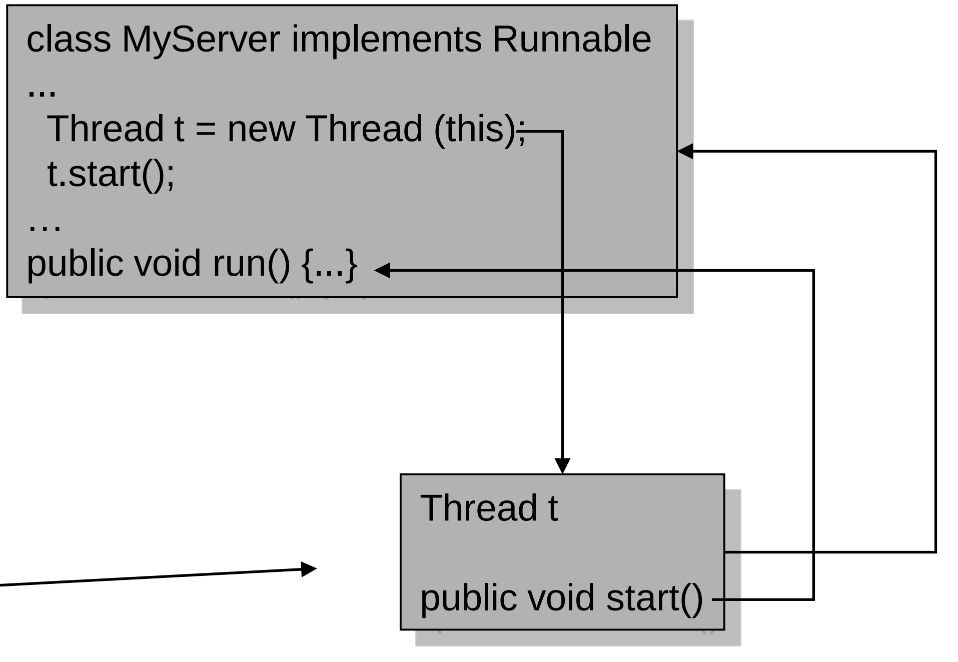
```
// java.lang.Runnable
// java.lang.Thread

class MyServer implements Runnable {

  // In interface Runnable:
  public void run () {
    ...
  }

  public void launch() {
    ...
    new Thread (this).start ();
    ...
  }
}
```

# Threads in Java

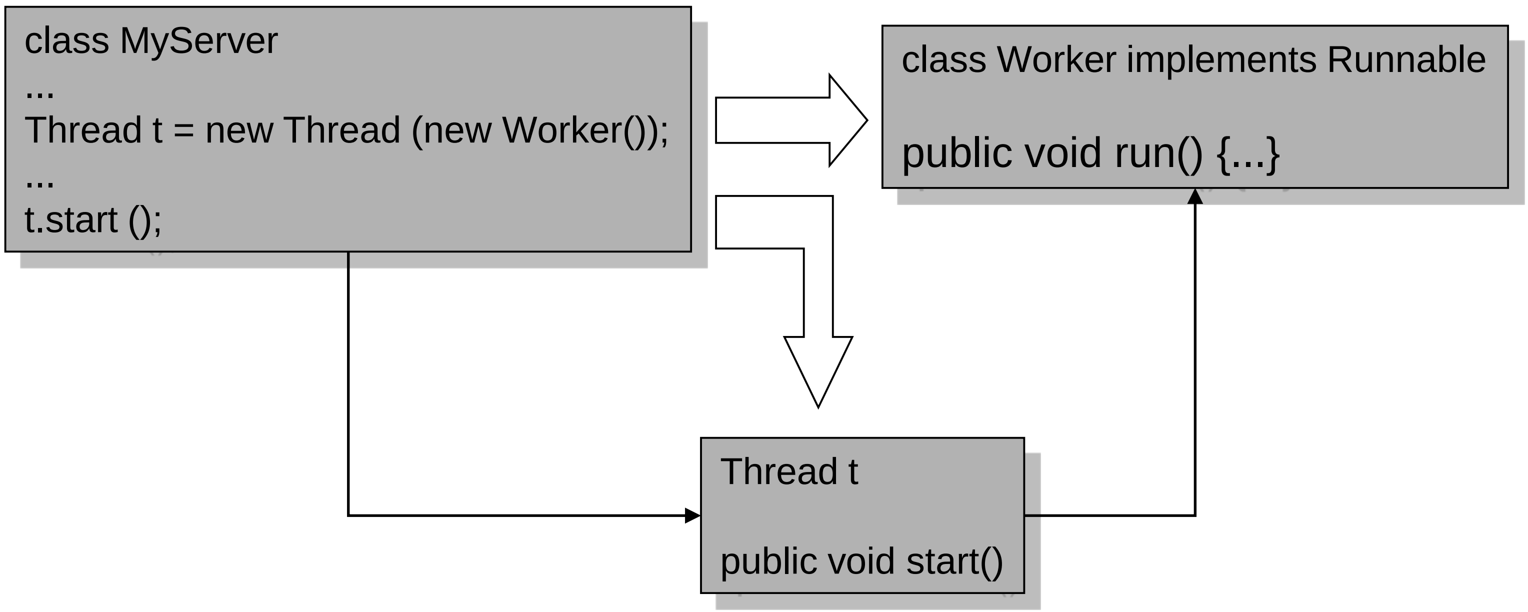
A class that supports threading implements interface Runnable.

Interface Runnable has one method: run(). This is a thread's entry point.

A new thread is created just like any other object. It is given the object where to execute and told to start running.



# Threads in Java




A thread is created to run in another object.
You cannot *start* a thread in static code, but
there are workarounds.

# Threads in Java

In Java you must use method Runnable.run(). Where is the diversity?

Programmatic or computed choice.

```
class MyServer implements Runnable {
  int h;

  // In interface Runnable:
  public void run () {
    switch (h) {
    case 1: foo(); break;
    case 2: bar(); break;
    }
  }

  public void main (String argv[]) {
    h = 1;
    new Thread (this).start ();
    h = 2;
    new Thread (this).start ();
  }
}
```

Race condition on variable h!

# Threads in Java

In Java you must use method Runnable.run(). Where is the diversity?

Programmatic or computed choice.

```
class MyServer implements Runnable {
  int h;

  // In interface Runnable:
  public void run () {
    switch (h) {
    case 1: foo(); break;
    case 2: bar(); break;
    }
  }

  public void main (String argv[]) {
    h = 1;
    new Thread (this).start ();
    h = 2;
    new Thread (this).start ();
  }
}
```

--NO--

Race condition on variable h!

# Threads in Java

In Java you must use method Runnable.run(). Where is the diversity?

Separate out methods into their own classes.

```
class Foo implements Runnable {
  public void run () {
    ...
  }
}
```

```
class Bar implements Runnable {
  public void run () {
    ...
  }
}
```

```
class MyServer {

  public void launch() {
    new Thread (new Foo ()).start ();
    new Thread (new Bar ()).start ();
  }
}
```

# Threads in Java

In Java you must use method Runnable.run(). Where is the diversity?

```
class MyServer {

  protected void foo() {
    ...
  }

  public void launch() {
    ...
    new Thread (new Runnable () {
      public void run () {
        foo();
      }
    }).start();
  }
}
```

Anonymous implementation

# Threads in Java

Wait for a thread to die: **Thread.join()**

```
class MyServer {

  public void main (String argv[]) {
    ...
    // Create thread t.
    Thread t = new Thread (new Foo ()).start ();
    ...
    // Main thread waits for t to die, i.e. return
    // from its call to Runnable.run()
    t.join ();
    ...
  }
}
```

# Threads in Java

`java.lang.Thread`

- `static void sleep (long millis)` – suspend caller for at least ms milliseconds
- `void join ()` – suspend caller until called thread has exited

`java.lang.Object`

- `void wait()` – suspend caller until notified*
- `void notify()` – release one waiting thread*
- `void notifyAll()` – release all waiting threads*

* The caller must own (have) the object's monitor

# Synchronization in Java

Threads that access  common variables together can seriously mess up the state of the program.

Synchronization is achieved by monitors.

A monitor is a non-sharable entity associated with every Java object instance.

A thread must have the monitor of the choosen object to be able to execute the code synchronized on that object.

# Synchronization in Java

When a method is declared as synchronized the monitor is retrieved from the method's object.

```
public synchronized void enqueue()
{...}
```

When a block of code is synchronized, any object's monitor can be used:

```
synchronized (myQueue) { ... }
```

# Synchronization in Java

A thread that attempts to enter a synchronized method or block must wait in a queue for the monitor.

When the monitor is released the thread continues to execute.

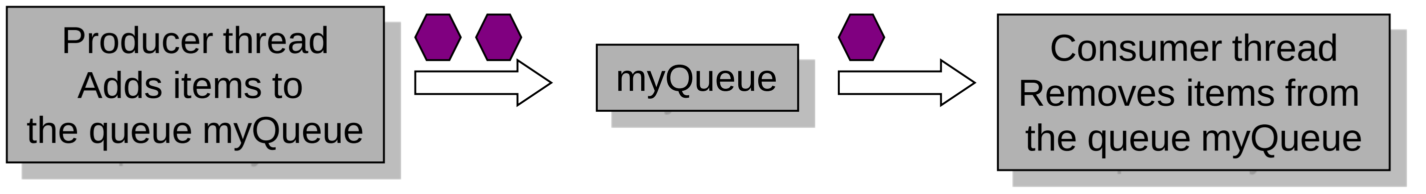
While the thread is inside the synchronized section it can release the monitor and go back to the waiting queue.

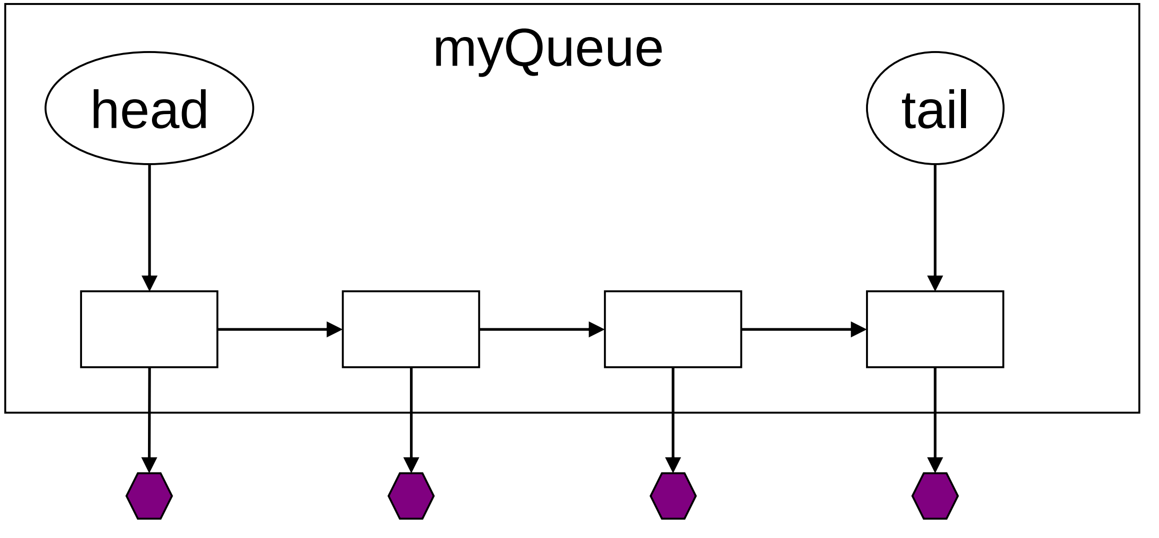
```
synchronized (myQueue) { ...
  myQueue.wait (); // Nothing to do
  ...}
```

When some other thread has the monitor it can *notify* a waiting thread, thus allowing the waiting thread to continue:

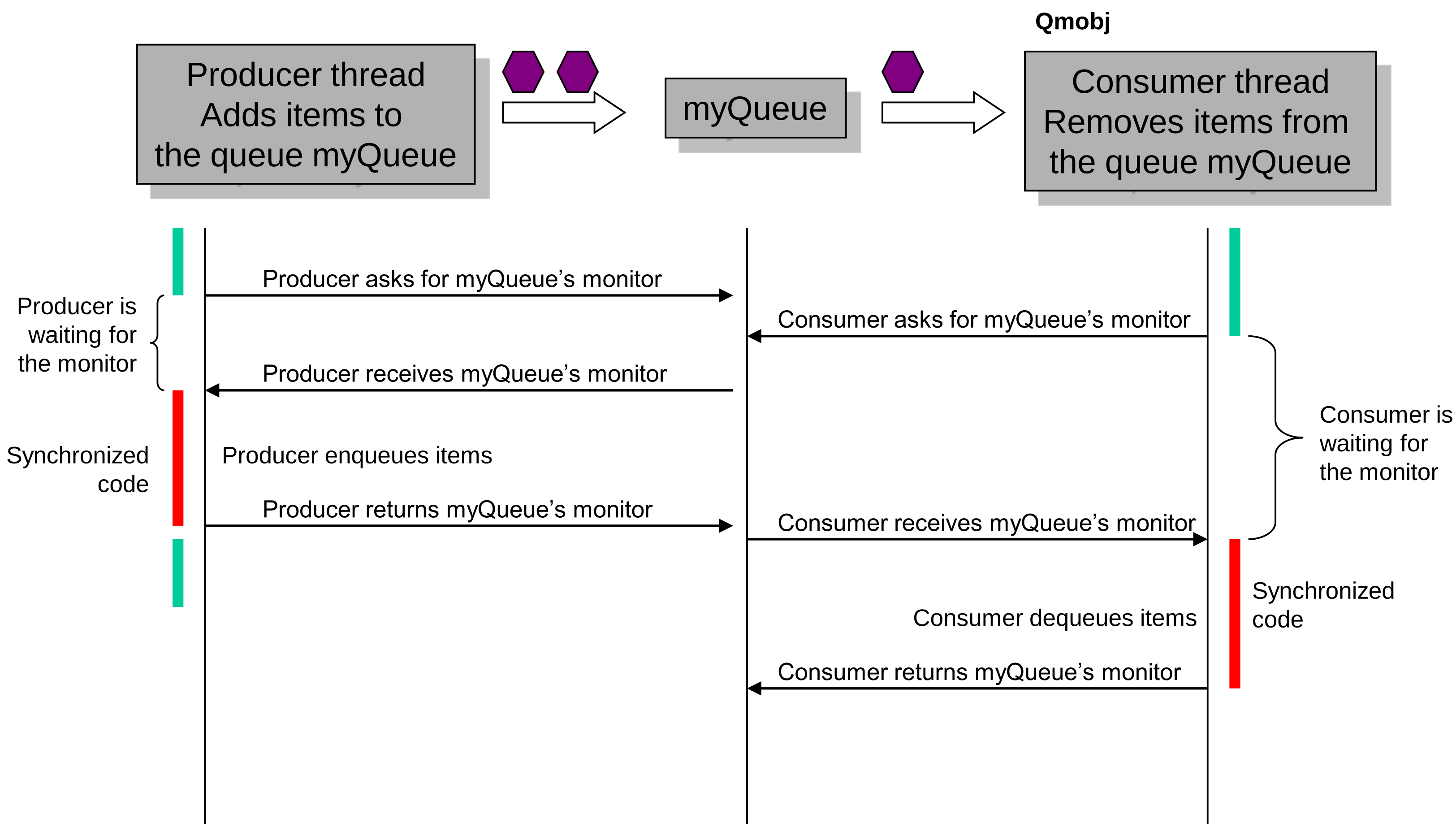
```
synchronized (myQueue) { ...
  myQueue.notify (); // Work is ready for you
  ...}
```

# Synchronization in Java

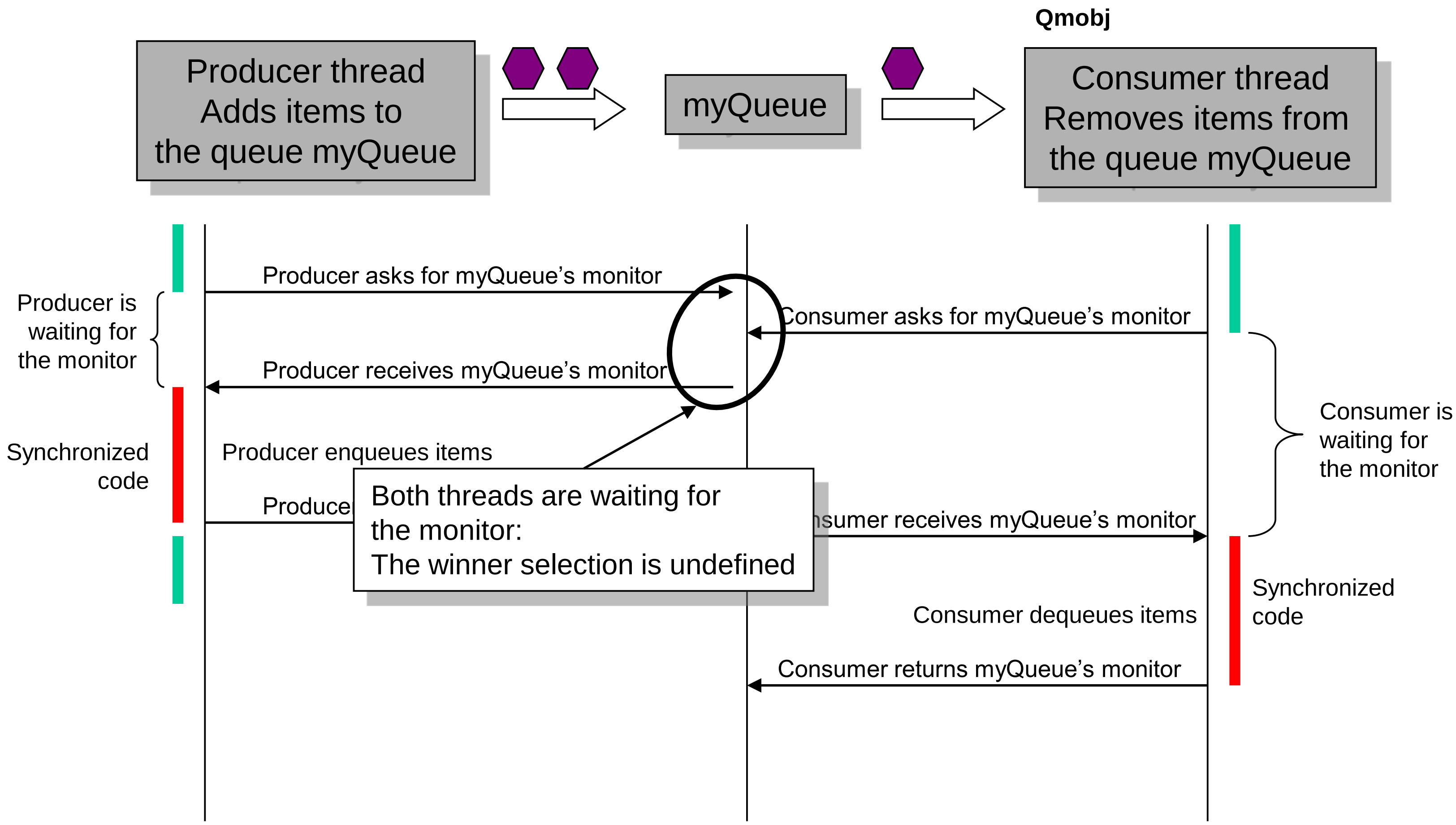


# Synchronization in Java



# Synchronization in Java



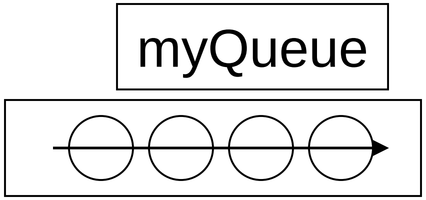
# Synchronization in Java

The wait()/notify() mechanism of a consumer-producer pair with a queue between them.

Access to the queue must be synchronized.

While the queue is empty the consumer does not execute.

```
// Producer
synchronized (myQueue) {
  myQueue.enqueue (x);
  myQueue.notify();
}
```




```
// Consumer
for (;;) {
  while (!myQueue.isEmpty()) {
    synchronized (myQueue) {
      e = myQueue.dequeue ();
    }
    process(e);
  }
  synchronized (myQueue) {
    myQueue.wait ();
  }
}
```

# Thread priorities

- `int Thread.getPriority()`
- `void Thread.setPriority(int newPriority)`

- Priority is a value from 1 (low) to 10 (high)
- Recommended constants:
- `Thread.MIN_PRIORITY        1`
- `Thread.NORM_PRIORITY       5`
- `Thread.MAX_PRIORITY       10`

# User threads vs daemon threads

- User threads do the actual work (high priority)
  - The JVM waits until all user threads have exited
  - User threads create user threads
- Daemon threads support user threads (low priority)
  - The JVM does not wait for daemon threads
  - Daemon threads create deamon threads

- `Thread thread = new Thread(...)`
- `thread.setDaemon(true); // before thread start`
- `thread.start();`

# Transient variables in Java

- `transient Object q = ...;`

- The transient keyword indicates a class variable that should not be serialized
- When the object is restored, the variable value is null and must be recreated if needed

Transient variables are related to serialization, not threads

# Volatile variables in Java

- `volatile int k = 0;`

- The volatile keyword instructs the compiler not to assume that the variable is updated by a single thread

- The compiler avoids certain optimizations

# java.util.concurrent.atomic

- Classes that provide thread-safe manipulation of single variables: boolean, integer, long, reference
- `boolean compareAndSet(expectedValue, updateValue)`
- If the variable equals expectedValue, it is assigned updateValue, and true is returned

# java.util.concurrent.atomic

- Classes that provide thread-safe manipulation of single variables: boolean, integer, long, reference

- Not a replacement for primitive datatypes
- Ties into hardware instructions where available
- Intended for non-blocking code

# Not mentioned ...

Avoid Thread. stop() suspend() resume() (deprecated)

Class Threadgroup : handle threads as a unit

Class ThreadLocal : thread-specific ‘global’ vars

Thread intercommunication with pipes.

java.util.concurrent.ThreadLocalRandom: non-shared random number generator

# End