# DETERMINISTIC DEFINITION OF CONCURRENT BEHAVIOR

ADVANCED SOFTWARE TOOLS SEMINAR

TEL AVIV UNIVERSITY
JULY 2012

# AGENDA

- motivation
  - problem
  - different approaches
- solution
  - idea
  - demo
  - related works
- conclusion
  - additional ideas
  - feedback

# MOTIVATION

during the last years software become more and more parallel

- multicore hardware
- new api's, libraries and frameworks
- new patterns and architectures

# MOTIVATION

- developing concurrent software is more complicated and challenging
  - synchronization
  - data races
- fortunately there are tools supporting development process
- but what about QA/UT?

# PROBLEM

- all modern QA/UT methodologies are based on one pillar:

executing the **same code** with the **same inputs** will result with the **same output**

is this true for concurrent code?

# POSSIBLE SOLUTIONS

- stress testing
- static analysis
- runtime analysis
- context switches randomization/enumeration
- different combinations of above techniques

# BUT …

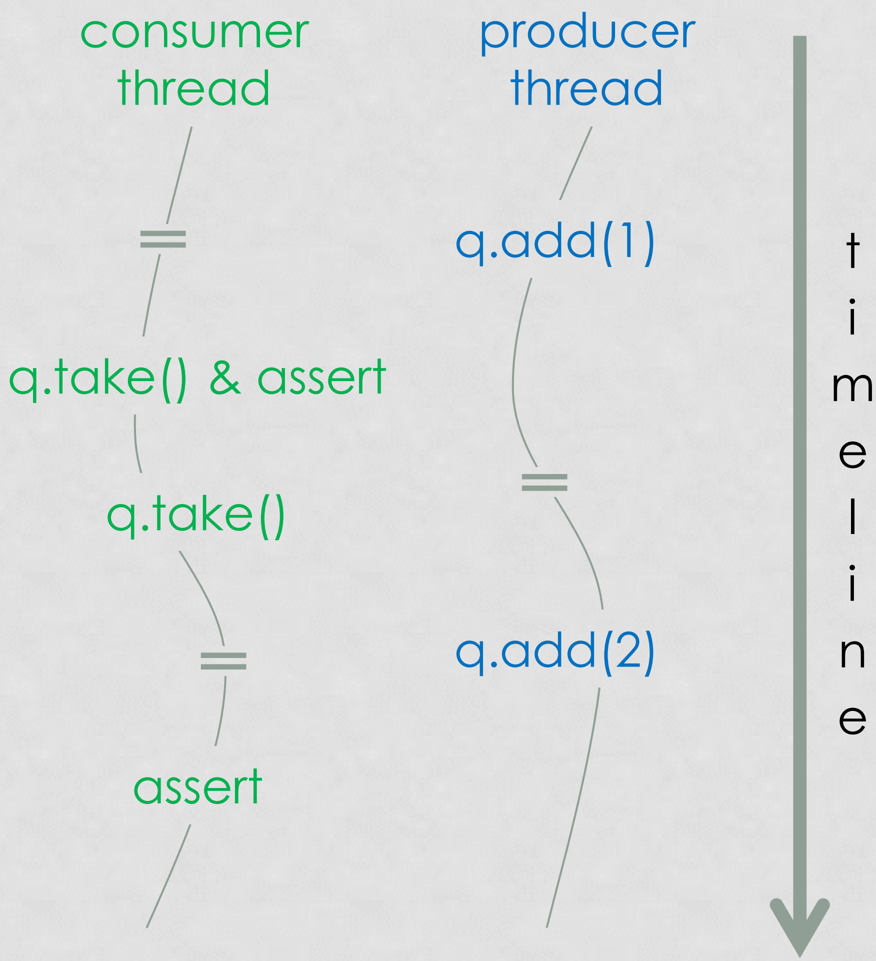
- pure performance
- inability to cover all possible scenarios
- false alarms / misses
- non deterministic
- deals with simple synchronization methods / scenarios only
- introduce new dedicated languages / notations
- requires dedicated runtime / source code modifications / instrumentation

# REAL LIFE

```
@Test
public void BlockingCollectionTests() throws Exception {

          final ArrayBlockingQueue<Integer> q = new ArrayBlockingQueue<Integer>(1);

          Thread addThread = new Thread(new Runnable() {
                    public void run() {

                              q.add(1);

                              try {
                                        Thread.sleep(100);
                              } catch (InterruptedException e) {
                                        e.printStackTrace();
                              }

                              q.add(2);
                    }
          });

          addThread.start();
          Thread.sleep(50);

          Integer taken = q.take();
          assertTrue(taken == 1 && q.isEmpty());

          taken = q.take();
          assertTrue(taken == 2 && q.isEmpty());

          addThread.join();
}
```

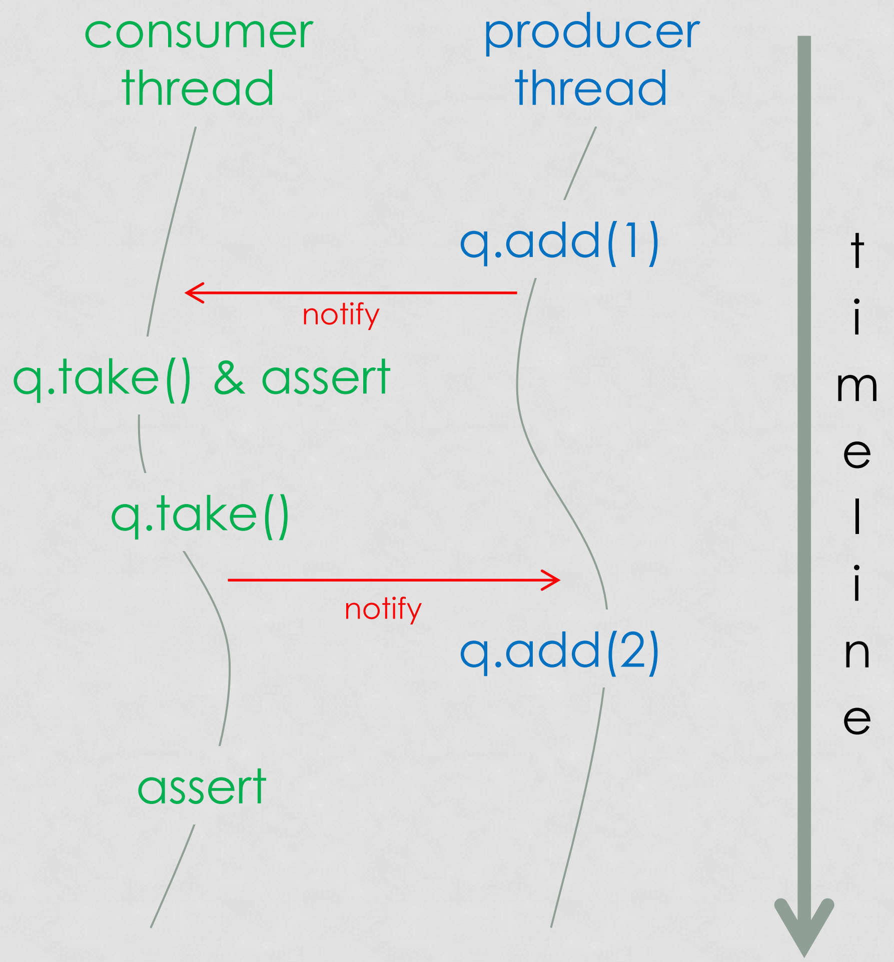
# REAL LIFE

```
Thread addThread = new Thread(new Runnable() {
    public void run() {

        q.add(1);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        q.add(2);
      }
});

addThread.start();
Thread.sleep(50);

Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());

taken = q.take();
assertTrue(taken == 2 && q.isEmpty());
```



# REAL LIFE

```
Thread addThread = new Thread(new
Runnable() {
   public void run() {

     q.add(1);
     add1.notify();

     take2.wait();
     q.add(2);
   }
});

addThread.start();

add1.wait();
Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());

taken = q.take();
take2.notify();

assertTrue(taken == 2 && q.isEmpty());
```



# IDEA

lets define new concept of Gate

$$G = \{ L , C \}$$

where:

- L – location in code
- C – boolean condition

when thread T reaches location L it is suspended until C becomes true

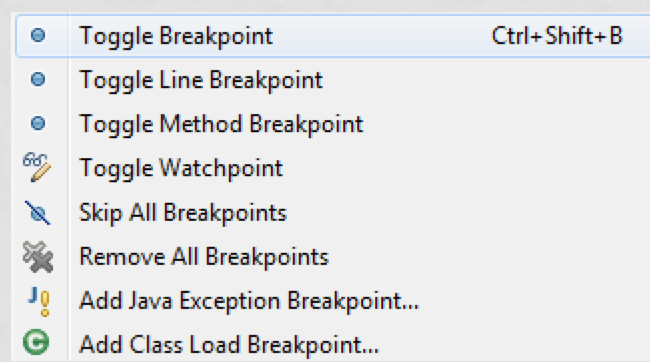
events are very simple implementation of gate

# IMPLEMENTATION

G = { L , C }

- C could be defined using standard Java syntax
- but what about L?
  - how we can define some location in the executable?
  - how we can intercept the execution to check the value of C / suspend the thread?
- the answer is very simple and it already exists in every modern platform / IDE

BREAKPOINT



# PUTTING THE THINGS TOGETHER

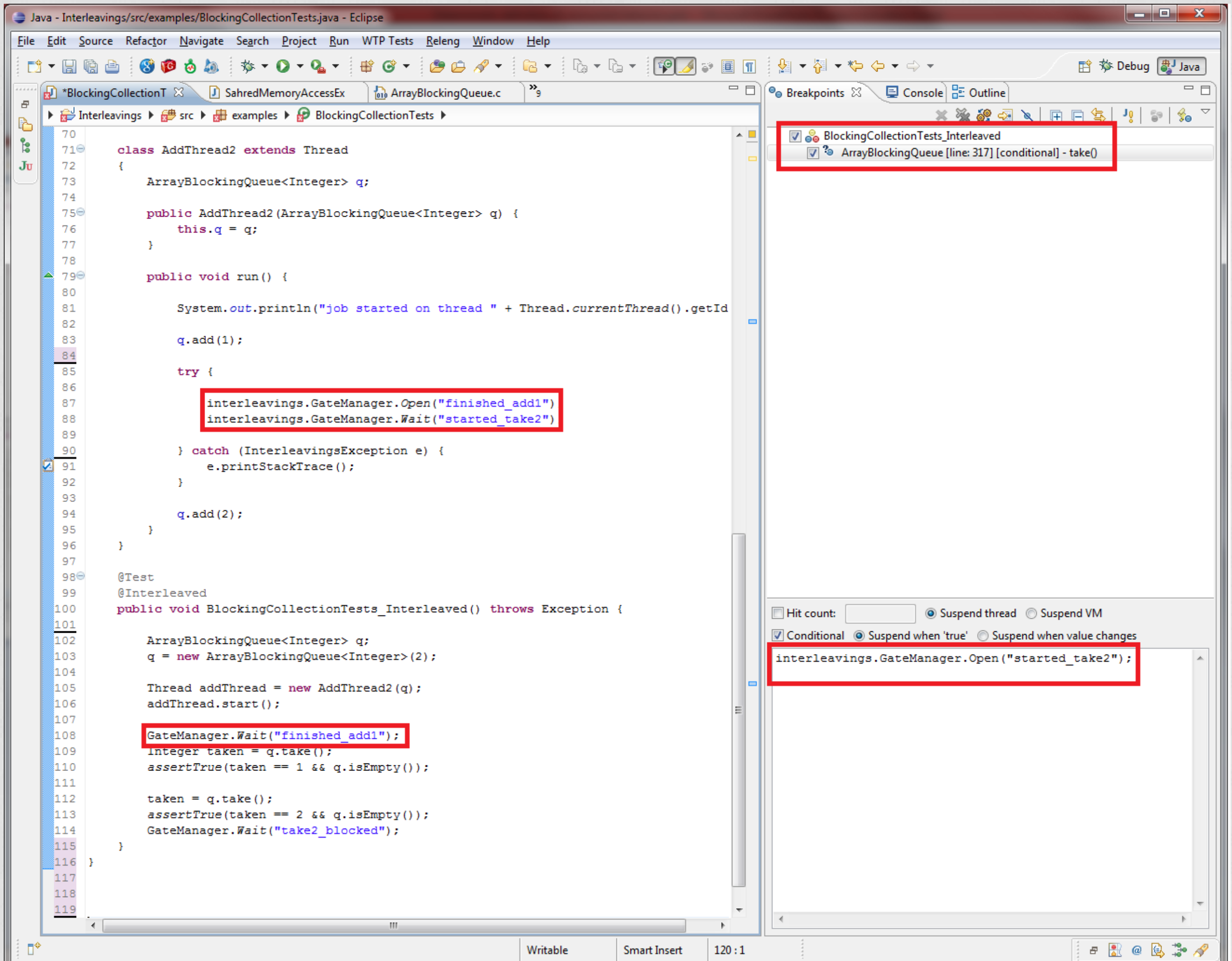
to define given thread scheduling we have to:

- define gates locations using breakpoints
- define gates conditions that will suspend/resume the threads

at runtime:

- the breakpoint will be hit
- the condition will be evaluated
- the thread will be suspended / resumed according to condition's value

# PUTTING THE THINGS TOGETHER



# QUESTIONS

# DEMOS

- shared memory access
- long running task
- first chance exception
- jobs collection
- blocking collection

# PROS

deterministic

- reproducible
- user defined scenarios
- allows to apply testing methodologies / tools to concurrent code

expressiveness

- fine control over gates locations (method, exception and conditional bp, hit counters, …)
- power of Java to define condition (interaction with local and private variables, method calls, …)
- allows to introduce more complex gates

# PROS

- allows to control third parties behaviors
- no CUT modifications / adaptations required
- removes synchronization logic from the test code
- the same test code could be used to test multiple concurrent scenarios
- no dedicated runtime / special version / binaries instrumentation required, the same binaries could be used in production
- based on simple and well know concepts all developers are familiar with, no dedicated syntax / language required
- good IDE integration
- not limited to some platform / language

# MULTITHREADED TC [2007]

- splits timeline for multiple logical “ticks”
- defines rules for advancing the clock
- test can wait for some tick

  or check which tick is it now

- good for simple ordering scenarios
- becomes tricky for more complex scenarios
- can handle blocking events only

```
//MultithreadedTC
public class TestTakeWithAdd
extends MultithreadedTest {
    ArrayBlockingQueue<Integer> q;
    @Override
    public void initialize() {
        q = new ArrayBlockingQueue<Integer>(1);
    }
    public void addThread() throws Exception {
        q.add(1);
        waitForTick(2);
        q.add(2);
    }

    public void takeThread() throws Exception {
        waitForTick(1);
        Integer taken = q.take();
        assertTrue(taken == 1 && q.isEmpty());
        taken = q.take();
        assertTick(2);
        assertTrue(taken == 2 && q.isEmpty());
    }
}
```

# IMUNIT [2011]

- allows to define events in test code
- for each test defines desired events ordering

- clear declarative notation
- good for simple ordering scenarios
- does not support complex events
- does not support complex orderings
- can not control CUT / third parties execution

```java
@Test //IMUnit
@Schedule("finishedAdd1->startingTake1, " +
          "[startingTake2]->startingAdd2")
public void testTakeWithAdd() {
    final ArrayBlockingQueue<Integer> q;
    q = new ArrayBlockingQueue<Integer>(1);

    Thread addThread = new Thread(
            new Runnable() {
                @Override
                public void run() {
                    q.add(1);
                    @Event("finishedAdd1");
                    @Event("startingAdd2");
                    q.add(2);
                }
            }, "addThread");
    addThread.start();
    @Event("startingTake1");
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    @Event("startingTake2");
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    addThread.join();
}
```

# WHAT'S NEXT

Testing:

- control execution flow
- inject mock objects

Validation:

- assert state invariants
- validate method input / output

Instrumentation:

- inject log / trace outputs
- save object state for future inspection

Aspects

# FEEDBACK