ETH Zürich
Chair of Software Engineering

# Java and C# in Depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

## Exercise Session – Week 8

# Quiz 1: What is printed? (Java)

```
class MyTask implements Runnable {
    public void run() {
        throw new RuntimeException("Help!");
    }
}

public static void main(String[] args) {
    try {
        (new Thread(new MyTask())).start();
        System.out.println("Everything is ok!");
    } catch (RuntimeException e) {
        System.out.println("Something went wrong...");
    }
}
```

«Everything is ok!

Exception in thread...: Help!»

Exceptions from other threads are not propagated to *main*

# Quiz 1: A C# solution

In C# you can use asynchronous delegates to propagate exceptions to the main thread:

```
static void MyTask() { throw new Exception("Help!"); }
delegate void MyTaskInvoker();
public static void Main() {
        try {
            MyTaskInvoker method = MyTask;
            IAsyncResult res = method.BeginInvoke(null, null);
            method.EndInvoke(res);
            // This doesn't work:
            // new Thread(MyTask).Start();
        } catch (Exception) {
            Console.WriteLine("Something went wrong");
        }
```

# Quiz 2: What happens (C#)?

```
static void MyTask() {
    try {
        ... // Some heavy work
    } catch { ...
    } finally {
        Console.WriteLine("Very important cleanup");
    }
}

public static void Main() {
    Thread t = new Thread(MyTask);
    t.IsBackground = true;
    t.Start();
    ...
    t.Interrupt();
}
```

*finally* block may not be executed: the main thread may exit before that and the application does not wait for background threads to finish

# Quiz 3: What can go wrong? (Java)

```
public walkUnderTheRain() {
    if(!isRaining) {
        try { wait(); }
        catch (InterruptedException e) {…}
    }
    System.out.println("Walking under the rain!");
}
```

Shared variable

Don’t expect that the first interrupt we get is the one we need: use *while* instead of *if*

To call *wait* the enclosing method must be synchronized

(otherwise *IllegalMonitorStateException* is thrown at runtime)

# Quiz 3: A C# solution: wait handles

```
static EventWaitHandle rain = new AutoResetEvent(false);

static void WalkUnderTheRain() {
    rain.WaitOne();
    Console.WriteLine("Walking under the rain!”);
}

public static void Main() {
    new Thread(WalkUnderTheRain).Start();
    Thread.Sleep(500);
    rain.Set();
}
```

# Quiz 4.a: What happens? (Java)

```
class MyTask implements Runnable {
    public void run() {
        while (true) { }
    }
}
public static void main(String[] args) {
    try {
        Thread t = new Thread(new MyTask());
        t.start();
        t.interrupt();
        t.join();
        System.out.println("t interrupted");
    } catch (InterruptedException e) {…}
}
```

*run* does not handle interrupts

this code will be never executed

# Quiz 4.a: How to handle interrupts?

1. Calling methods that throw *InterruptedException*

```
public synchronized void run() {
    while (true) try {
        sleep (200);
    } catch (InterruptedException e) {
        return;
    }
}
```

2. Checking Thread.interrupted flag

```
public void run() {
    while (true) {
        if (Thread.interrupted()) { return; }
    }
}
```

# Quiz 4.b: What happens? (C#)

```
static void Run() { while (true) { } }

public static void Main() {
    Thread t = new Thread(Run);
    t.Start();
    Thread.Sleep(500);
    t.Abort();
    t.Join();
    Console.WriteLine("t aborted");
}
```

This code is executed.

Unlike *Interrupt*, *Abort* stops the thread even if it's currently running

# Quiz 4.c: What happens (C#)?

```
static void Run() {
    while (true) {
        try {
            Thread.Sleep(1000);
        } catch (ThreadAbortException e) {
            Console.WriteLine("Ha-ha! I will be executing FOREVER!");
        }
    }
}

public static void Main() {
    Thread t = new Thread(Run);
    t.Start();
    Thread.Sleep(500);
    t.Abort();
    t.Join();
    Console.WriteLine("t aborted");
}
```

Thread *t* is still aborted!

*ThreadAbortException* is automatically rethrown at the end of the *catch* block if *Thread.ResetAbort* is not called

# Quiz 5: Is this class thread-safe? (Java)



```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

```
Counter count = new Counter;
...
// In thread 1:
count.increment();
...
// In thread 2:
count.increment();
...
// In the main thread after joining
// threads 1 and 2:
System.out.println(count.value());
```

*c++* is not atomic =>

the result might be 1

# Quiz 5: Is this class thread-safe? (Java)

All attributes must be accessible only through synchronized methods

```
class Counter {
  private int c = 0;

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

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

# Quiz 5: Is this class thread-safe? (Java)

```
class Counter {
    ... // Everything as before

    public static synchronized void increment_other(Counter other) {
        other.c++;
    }

    Counter(int c) {
        this.c = c;
    }
}
```

No: static methods use the class as a lock!

OK: constructors need not (and cannot) be synchronized, they are executed once per object

# Quiz 6: What is printed? (Java)

```
public class Test extends Thread {
  boolean keepRunning = true;

  public static void main(String[] args) {
    Test t = new Test(); t.start();
    Thread.sleep(1000);
    t.keepRunning = false;
    System.out.println("keepRunning is false");
  }

  public void run() {
    while (keepRunning) {}
    System.out.println("finished");
  }
}
```

Fix by declaring attributes **volatile**

Thread might cache values locally. Here, it will run forever!

# Quiz 7: Does it work? (C#)

```
volatile static bool go;
volatile static DateTime dt;

static void Wait() {
        while (!go) { }
        Console.WriteLine(dt);
    }

public static void Main() {
   new Thread(Wait).Start();
   Thread.Sleep(1000);
    dt = DateTime.Now;
    go = true;
}
```

Here we want to see the change to *dt* made by the main thread

Compilation error:

Objects of non-primitive value types cannot be cached by the processor => need not (and cannot) be *volatile*

# C# volatile

- Only (up to) 32bit types can be declared volatile.
  - Reference types (just the reference is volatile)
  - sbyte, byte, short, ushort, int, uint, char, float, bool
- Threads will always get the most up-to-date value for volatile fields.
- Fields declared as volatile are not cached.

# Quiz 8: Communication via Mutex (C#)

- Given:
  - 1 Mutex
  - 2 Threads that can access only that Mutex

- How can you transfer data from one thread to the other, using ONLY the Mutex as a communication.

# Quiz 8: Communication via Mutex (C#)

| | |
|---|---|
| ReleaseMutex | Releases the Mutex once. |
| WaitOne() | Blocks the current thread until the current WaitHandle receives a signal. (Inherited from WaitHandle.)<br><br>Return Value: true if the current instance receives a signal. If the current instance is never signaled, WaitOne never returns. |
| WaitOne(Int32) | Blocks the current thread until the current WaitHandle receives a signal, using a 32-bit signed integer to specify the time interval. (Inherited from WaitHandle.)<br><br>Return Value: true if the current instance receives a signal; otherwise, false. |

# Quiz 8: Communication via Mutex (C#)

```
void SendData(int data)
{
  for (int i = 0; i < 32; i++) {
    if (((data >> i) & 0x1) == 1) {
      mutex.WaitOne();
      Thread.Sleep(timeout);
      mutex.ReleaseMutex();
    } else {
      Thread.Sleep(timeout);
    }
  }
}
```

```
void ReceiveData()
{
  for (int i = 0; i < 32; i++)
{
    if (mutex.WaitOne(0)) {
      mutex.ReleaseMutex();
      // bit is 0
    } else {
      // bit is 1
    }
    Thread.Sleep(timeout);
  }
}
```

# Questions?