Advanced Programming @ MEIC

(Lesson #18)

Last class

- Data races
- Java Memory Model
  - No out-of-thin-air values
  - Data-race free programs behave as expected

Today

- Finish with the Java Memory Model
- Introduction to concurrent objects
  - How to implement them?

Java Memory Model primitives

- synchronized blocks
- volatile fields
- final fields

Java Memory Model primitives

- synchronized blocks
- volatile fields
- **final fields**

Thread-safety and Immutable Objects

João Cachopo

INSTITUTO
SUPERIOR
TÉCNICO

java.util.Hashtable

vs

java.util.HashMap

What about

j.u.c.ConcurrentHashMap

?

What is thread-safety?

A piece of code is **thread-safe** if it can be used in a multi-threaded environment

-- Wikipedia

An object is **thread-safe** if it can be used in a multi-threaded environment

if it can be used?

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }
   public void getCount() {
      return count;
   }
}
```

Can this be used in a multi-threaded environment?

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }
   public void getCount() {
      return count;
   }
}
```

Yes, but not with correct behavior...

When can we say that a concurrent object is **correct**?

When can we say that a **sequential** object is correct?

Operations of a **correct** sequential object behave in a certain way

For instance, incrementing the counter twice and asking the count value should return 2

Does the same happen if the counter is used by multiple threads?

```
public class Counter {
  private int count = 0;

  public void inc() {
    count++;
  }
  public void getCount() {
    return count;
  }
}
```

Other problems may happen:
e.g., object's invariants may break

We shall return to this later...

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

How to fix this?

```
public class Counter {
   private volatile int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

Does this work?

```
public class Counter {
   private volatile int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

volatile ≠ inc atomic

```
public class Counter {
   private int count = 0;

   public synchronized void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

What about this?

```
public class Counter {
   private int count = 0;

   public synchronized void inc() {
      count++;
   }
   // and this one needs it too?
   public void getCount() {
      return count;
   }
}
```

Do we have a data-race?

YES

- Thread T1 reads count
- Thread T2 writes to count
- No sync among them

```
public class Counter {
   private int count = 0;

   public synchronized void inc() {
      count++;
   }
   // and this one needs it too?
   public void getCount() {
      return count;
   }
}
```

Does it matter?

If we want linearizability, yes!

(Without sync, a thread repeatedly calling count may always get the same value, even if inc was called many times by other threads)

```
public class Counter {
   private volatile int count = 0;

   public synchronized void inc() {
      count++;
   }
   // and this one needs it too?
   public void getCount() {
      return count;
   }
}
```

Does this solve it?

```
public class Counter {
   private volatile int count = 0;

   public synchronized void inc() {
      count++;
   }
   // and this one needs it too?
   public void getCount() {
      return count;
   }
}
```

Yes! No data-races.

Immutable objects
are thread-safe

(even if they are shared through a data-race!)

What is an immutable object?

Informally:
An immutable object is an object
whose **state cannot be changed**
after the object is created

How to create immutable objects in Java?

Don’t allow them to change!

```
public class Date {
   int day;
   int month;
   int year;

   public Date(int d, int m, int y) {
      day = d;
      month = m;
      year = y;
   }
}
```

Is this OK?

```
Date d = new Date(1, 4, 2011);
sharedVar = d;
d.day = 3;
```

```
public class Date {
   int day;
   int month;
   int year;

   public Date(int d, int m, int y) {
      day = d;
      month = m;
      year = y;
   }
}
```

How to fix this?

```
public class Date {
   private int day;
   private int month;
   private int year;

   public Date(int d, int m, int y) { ... }
   public int day() { return day; }
   public int month() { return month; }
   public int year() { return year; }
   // NOTE: no setters!
}
```

Is it OK now?

```
Date d = new Date(1, 4, 2011);
sharedVar = d;   // normal write (data-race)


// on another thread
Date d = sharedVar; //normal read (data-race)
if (d != null) {     //saw the new date object?
   int year = d.year();
}
```

Is there a problem?

```
Date d = new Date(1, 4, 2011);
sharedVar = d;   // normal write (data-race)


// on another thread
Date d = sharedVar; //normal read (data-race)
if (d != null) {     //saw the new date object?
   int year = d.year();   //may return 0
}
```

Is there a problem?

```
public class Date {
  private int day;
  private int month;
  private int year;

  public Date(int d, int m, int y) { ... }
  public int day() { return day; }
  public int month() { return month; }
  public int year() { return year; }
  // NOTE: no setters!
}
```

How to fix it?

```
public class Date {
  private final int day;
  private final int month;
  private final int year;

  public Date(int d, int m, int y) { ... }
  public int day() { return day; }
  public int month() { return month; }
  public int year() { return year; }
  // NOTE: no setters!
}
```

The JMM to the rescue!

```
Date d = new Date(1, 4, 2011);
sharedVar = d;  // normal write (data-race)


// on another thread
Date d = sharedVar; //normal read (data-race)
if (d != null) {     //saw the new date object?
   int year = d.year();   //always returns 2011
}
```

Using `final` is not enough...

```
public class Date {
   private final int day;
   private final int month;
   private final int year;

   public Date(int d, int m, int y) {
      day = d;
      month = m;
      year = y;
   }
   // getters...
}
```

```
public class Date {
   private final int day;
   private final int month;
   private final int year;

   public Date(int d, int m, int y) {
      day = d;
      month = m;
      year = y;
      sharedVar = this;   // let this escape
   }
   // getters...
}
```

```
Date d = new Date(1, 4, 2011);
```

```
Date d = new Date(1, 4, 2011);


// on another thread
Date d = sharedVar; //normal read (data-race)
if (d != null) {     //saw the new date object?
   int year = d.year();   //may return 0
}
```

The `final` guarantees work only after the constructor finishes

One last note about the `final` semantics

```
public class Foo {
   public int x;  // no final here
   public Foo(int x) { this.x = x; }
}

public class Bar {
   private final Foo f;

   public Bar() {
      this.f = new Foo(3);
   }
   public Foo foo() { return f; }
}
```

```
sharedVar = new Bar();   //normal write (data-race)


// on another thread
Bar b = sharedVar; //normal read (data-race)
if (b != null) {     //saw the new bar object?
   int x = b.foo().x;   // ??
}
```

```
sharedVar = new Bar();   //normal write (data-race)


// on another thread
Bar b = sharedVar; //normal read (data-race)
if (b != null) {     //saw the new bar object?
   int x = b.foo().x;   //always returns 3
}
```

Other threads will see versions of any object or array referenced by final fields that are at least as up-to-date as the final fields are

References

- The Java Language Specification, Section 17.5: Final Field Semantics
- Immutable Objects (The Java™ Tutorials > Essential Classes > Concurrency)

Concurrent objects

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

```
public class Counter {
   private int count = 0;

   public synchronized void inc() {
      count++;
   }

   public synchronized void getCount() {
      return count;
   }
}
```

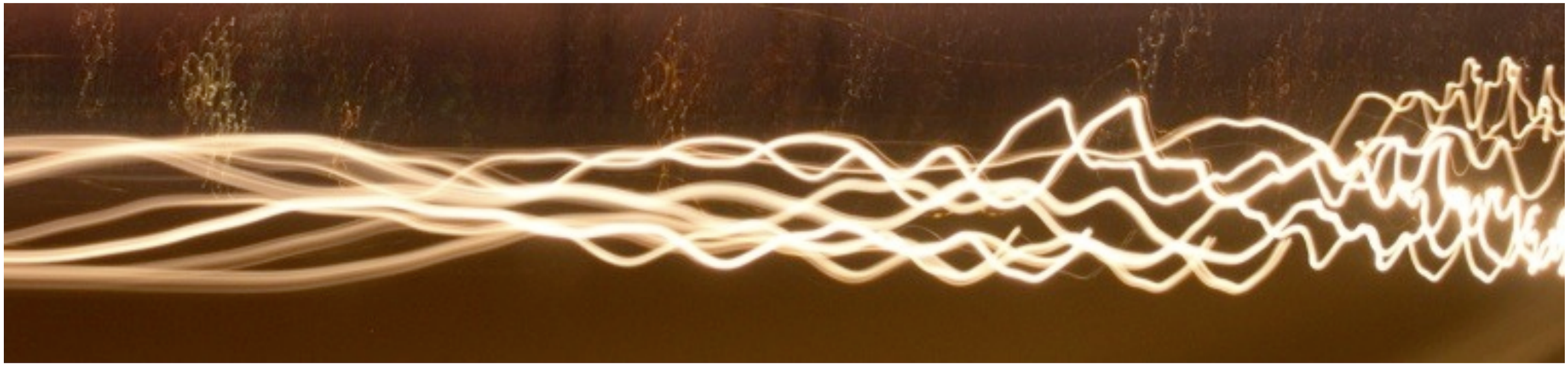
```
public class Counter {
   private int count = 0;

   public synchronized void inc() {
      count++;
   }

   public synchronized void getCount() {
      return count;
   }
}
```

Is this the only way?

Using a Software Transactional Memory



JVSTM

Software Transactional Memory as a Java library

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
class Person {
  String name;

  String getName() {
    return name;
  }

  void setName(String name) {
    this.name = name;
  }
}
```

$\Rightarrow$

```
class Person {
  final VBox<String> name
    = new VBox<String>(null);

  String getName() {
    return name.get();
  }

  void setName(String name) {
    this.name.put(name);
  }
}
```

```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

Must be immutable!

```
class Person {
  String name;

  String getName() {
    return name;
  }

  void setName(String name) {
    this.name = name;
  }
}
```

⇒

```
class Person {
  final VBox<String> name
    = new VBox<String>(null);

  String getName() {
    return name.get();
  }

  void setName(String name) {
    this.name.put(name);
  }
}
```

```
public class Transaction {
   public static void start();
   public static void abort();
   public static void commit();
}
```

```
public class Transaction {
   public static void start();
   public static void abort();
   public static void commit();
}
```

```
class SomeClass {
   @Atomic
   void swapNames(Person p1, Person p2) {
      String tmp = p1.getName();
      p1.setName(p2.getName);
      p2.setName(tmp);
   }
}
```

```
public class Transaction {
   public static void start();
   public static void abort();
   public static void commit();
}
```

You should be using this

```
class SomeClass {
   @Atomic
   void swapNames(Person p1, Person p2) {
      String tmp = p1.getName();
      p1.setName(p2.getName);
      p2.setName(tmp);
   }
}
```

If using the `@Atomic` annotation:

```
java jvstm.ProcessAtomicAnnotations \
     <directory with all .class files>
```

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

```
public class Counter {
   private final VBox<Integer> count =
                 new VBox<Integer>(0);

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

```
public class Counter {
   private final VBox<Integer> count =
                 new VBox<Integer>(0);

   public void inc() {
      count.put(count.get() + 1);
   }

   public void getCount() {
      return count.get();
   }
}
```

```
public class Counter {
   private final VBox<Integer> count =
                 new VBox<Integer>(0);

   @Atomic public void inc() {
      count.put(count.get() + 1);
   }

   public void getCount() {
      return count.get();
   }
}
```

First **multi-version** STM

Designed for **very large** transactions
and **high read/write ratio**

It is a **lock-free** STM

How to implement **atomic** operations **without locks**?

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }

   public void getCount() {
      return count;
   }
}
```

```
public class Counter {
   private int count = 0;

   public void inc() {
      count++;
   }

      . . .
}
```

```
public class Counter {
   private int count = 0;

   public void inc() {
      int c = count;
      int newC = c + 1;
      count = newC;
   }

   . . .
}
```

```
public class Counter {
   private int count = 0;

   public void inc() {
      int c = count;     // read
      int newC = c + 1; // do something
      count = newC;      // update (write)
   }

   ...
}
```

```
public class Counter {
   private int count = 0;

   public void inc() {
      int c = count;     // read
      int newC = c + 1; // do something
      count = newC;      // update (write)
   }

   . . .
}
```

Only if count was not changed by another thread...

```
public class Counter {
   private int count = 0;

   public void inc() {
      int c = count;
      int newC = c + 1;
      CAS(count, c, newC);
   }

   . . .
}
```

CAS = compare-and-set

```
CAS(var, old, new) ≈

  if (var == old) {
    var = new;
    return true;
  } else {
    return false;
  }
```

```
CAS(var, old, new) ≈

  if (var == old) {
    var = new;
    return true;
  } else {
    return false;
  }
```

But, it is atomic!

```
public class Counter {
   private int count = 0;

   public void inc() {
      int c = count;
      int newC = c + 1;
      CAS(count, c, newC);
   }

   . . .
}
```

What's the problem?

```
public class Counter {
  private int count = 0;

  public void inc() {
    while (true) {
      int c = count;
      int newC = c + 1;
      if CAS(count, c, newC) return;
    }
  }

  ...
}
```

```
public class Counter {
  private int count = 0;

  public void inc() {
    while (true) {
      int c = count;
      int newC = c + 1;
      if CAS(count, c, newC) return;
    }
  }

  ...
}
```

We can still do better...