

The Nonlinearity of `volatile` in Java (short talk)

John Boyland

University of Wisconsin-
Milwaukee

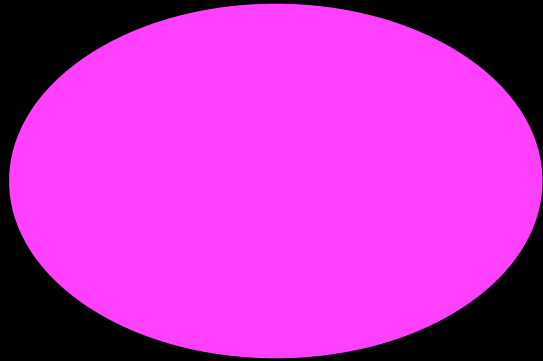
IWACO '08

Sampling Data (synch)

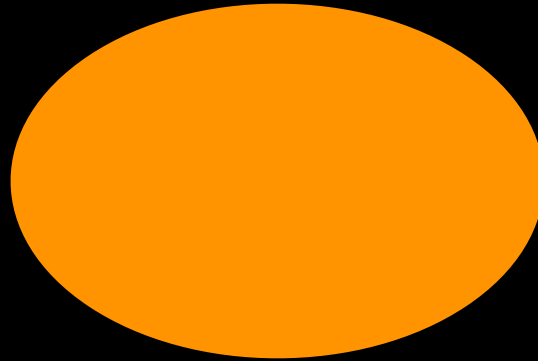
```
class Sample {  
    private Datum datum;  
    public synchronized  
    void push(int x, int y) {  
        datum = new Datum(x,y);  
    }  
    public synchronized Datum poll() {  
        Datum d = datum;  
        datum = null;  
        return d;  
    }  
}
```

Transfer (with synch)

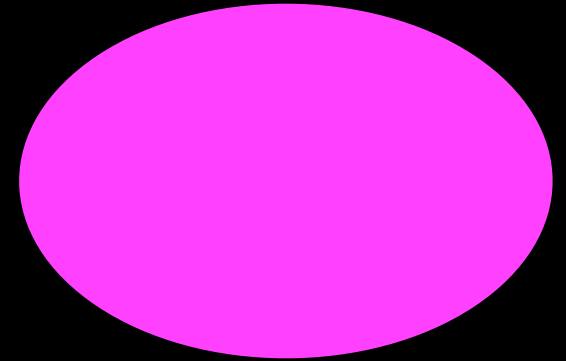
Producer



Sample



Consumer

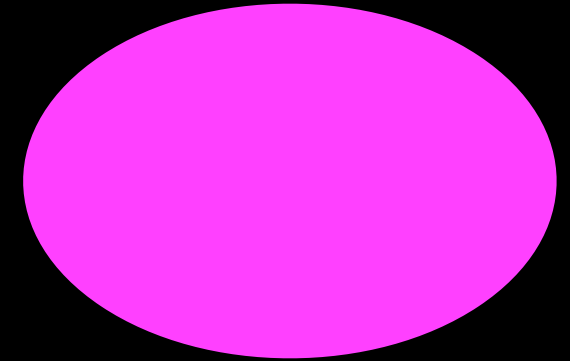
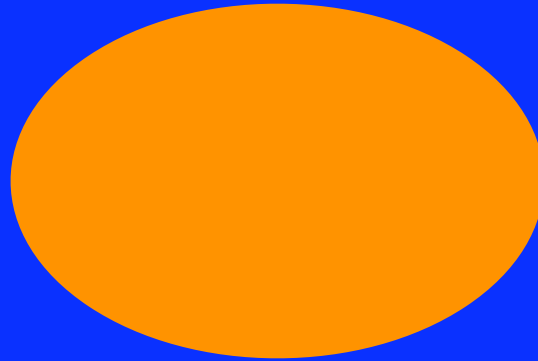
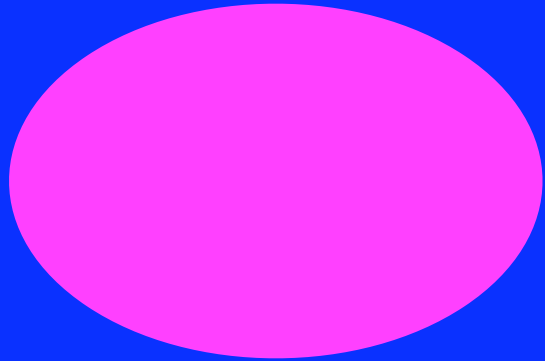


Transfer (with synch)

Producer

Sample

Consumer

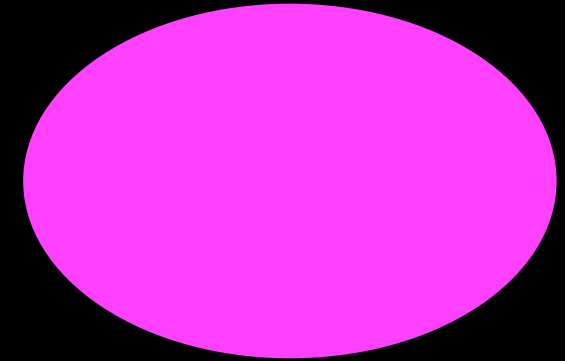
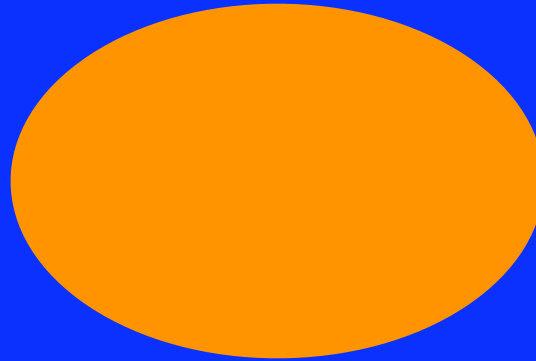
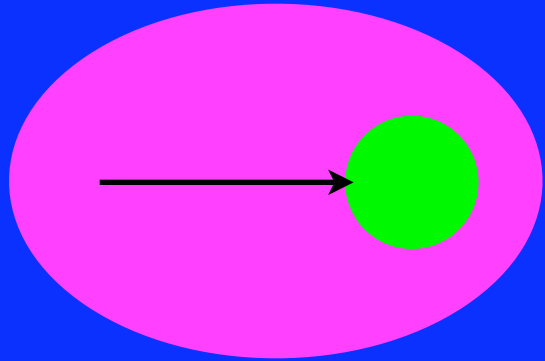


Transfer (with synch)

Producer

Sample

Consumer

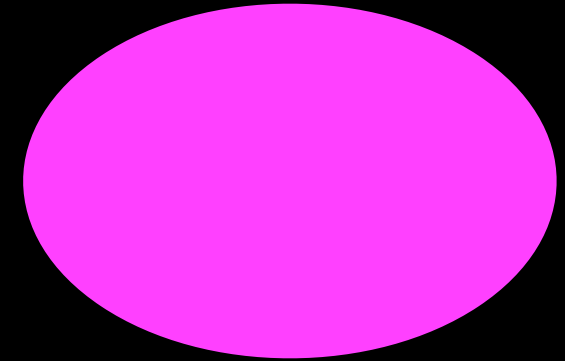
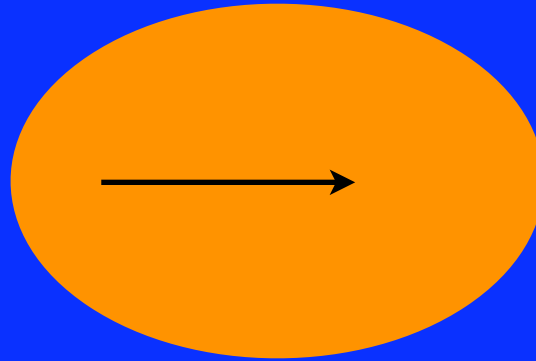
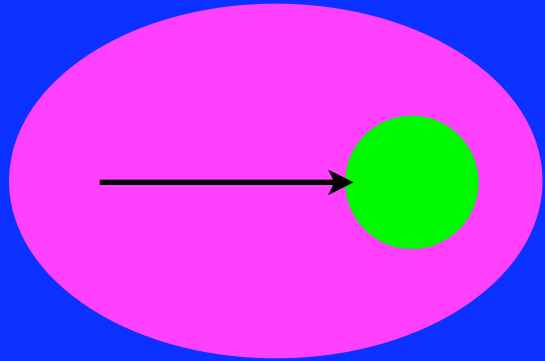


Transfer (with synch)

Producer

Sample

Consumer

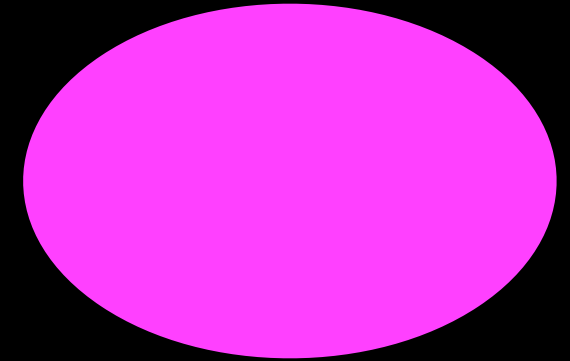
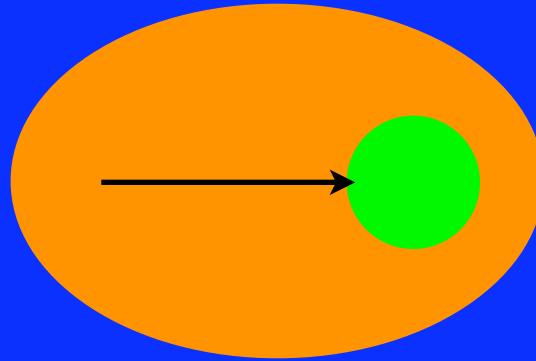
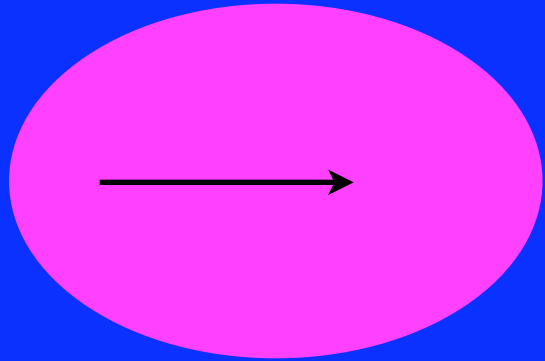


Transfer (with synch)

Producer

Sample

Consumer

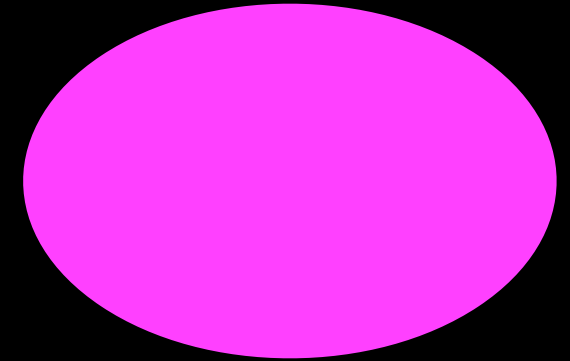
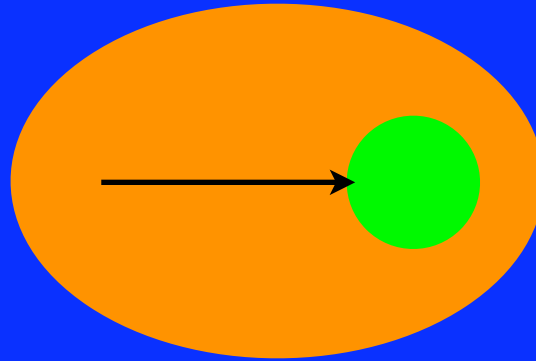
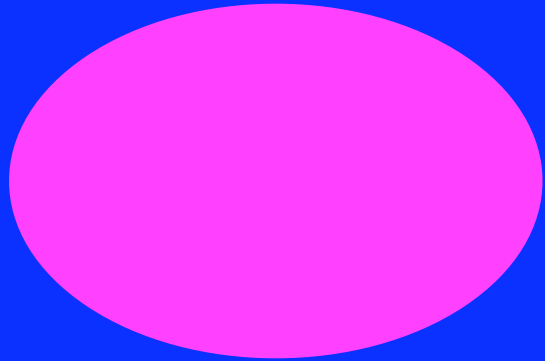


Transfer (with synch)

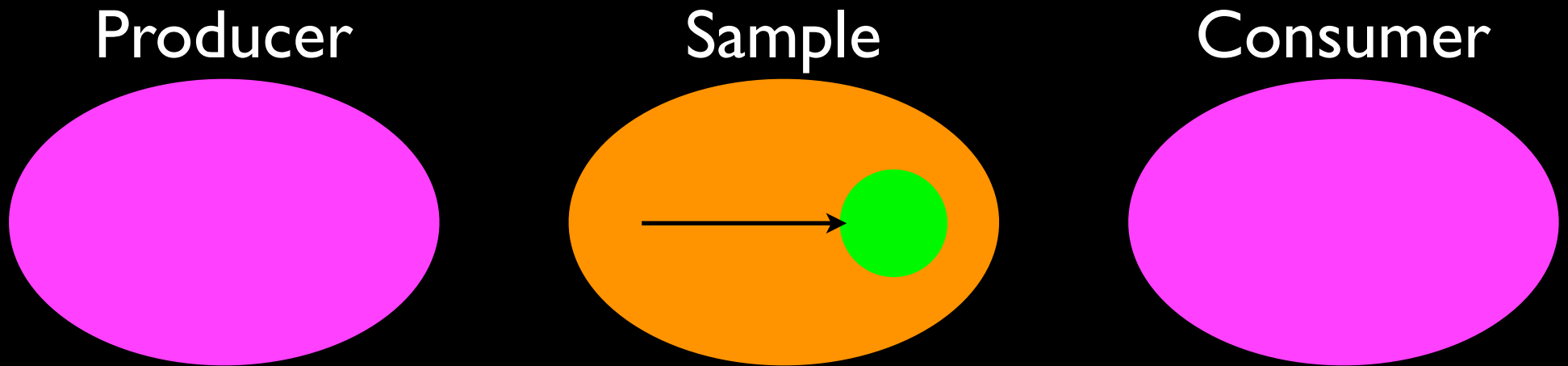
Producer

Sample

Consumer

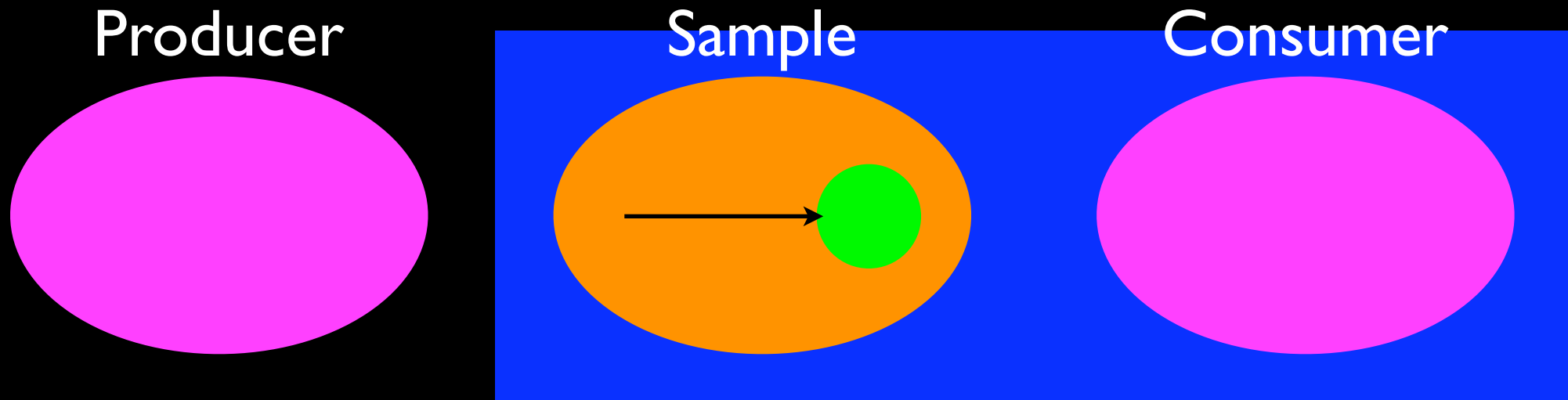


Transfer (with synch)



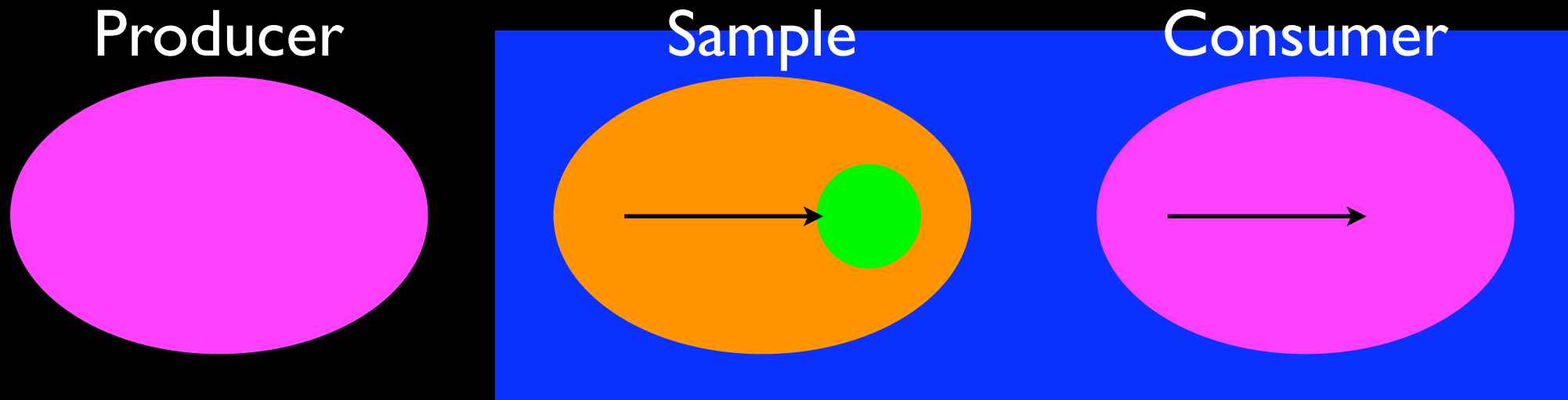
Sample Invariant: unless null, it includes permission

Transfer (with synch)



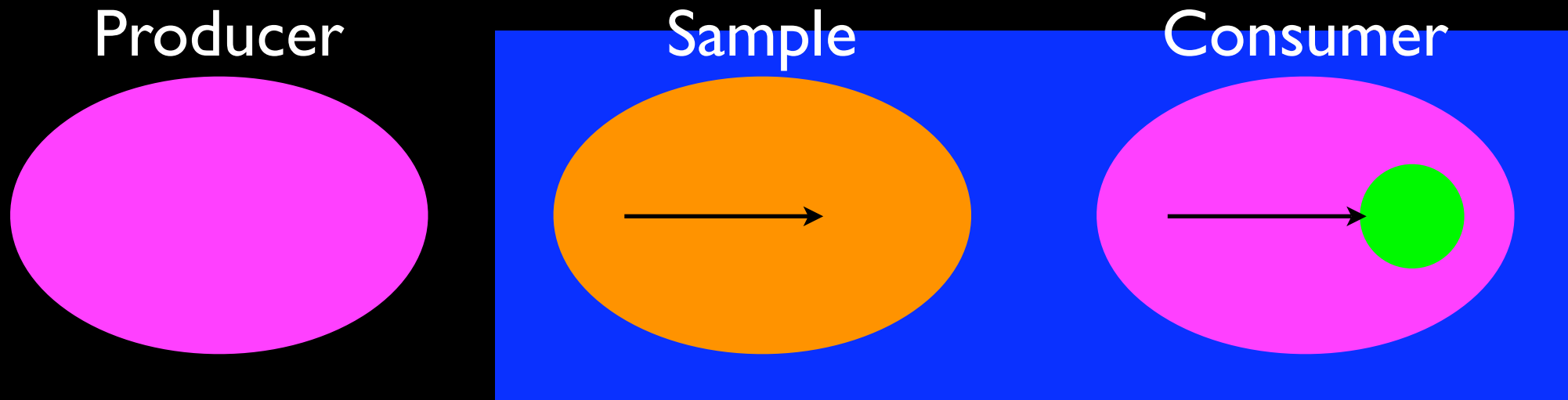
Sample Invariant: unless null, it includes permission

Transfer (with synch)



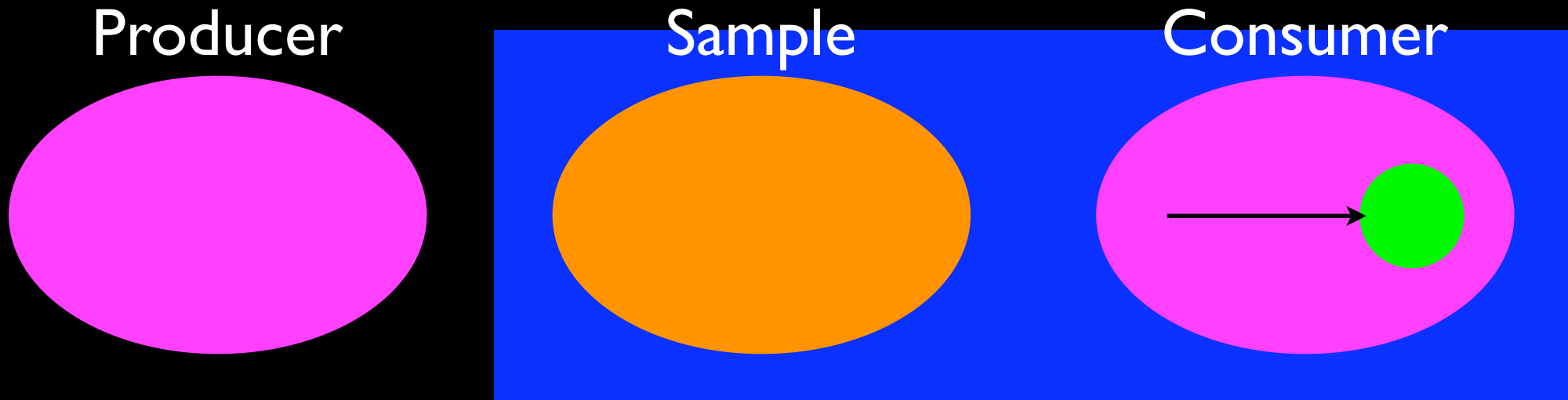
Sample Invariant: unless null, it includes permission

Transfer (with synch)



Sample Invariant: unless null, it includes permission

Transfer (with synch)



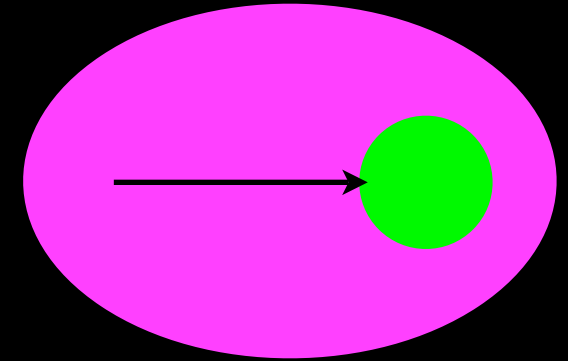
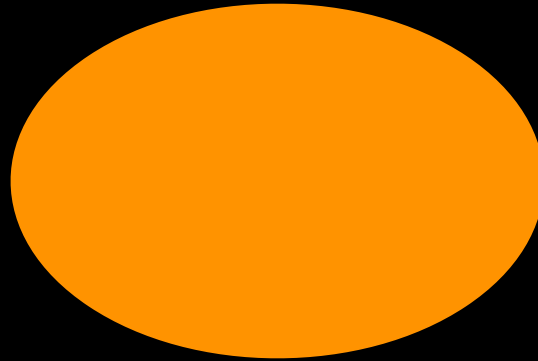
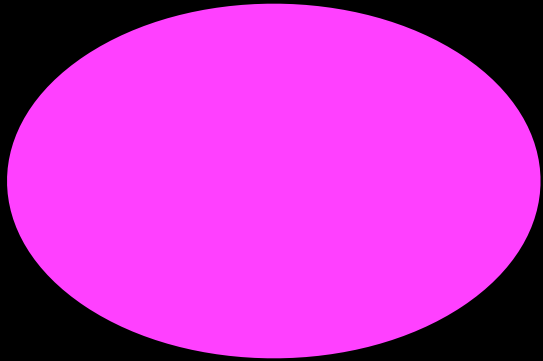
Sample Invariant: unless null, it includes permission

Transfer (with synch)

Producer

Sample

Consumer



Sample Invariant: unless null, it includes permission

Sample (volatile I)

```
class Sample {  
    private volatile Datum datum;  
    public void push(int x, int y) {  
        datum = new Datum(x,y);  
    }  
    public Datum poll() {  
        Datum d = datum;  
        datum = null;  
        return d;  
    }  
}
```

Sample (volatile I)

```
class Sample {  
    private volatile Datum datum;  
    public void push(int x, int y) {  
        datum = new Datum(x,y);  
    }  
    public Datum poll() {  
        Datum d = datum;  
        datum = null;  
        return d;  
    }  
}
```

Other consumers
may get same datum

Sample (volatile I)

```
class Sample {  
    private volatile Datum datum;  
    public void push(int x, int y) {  
        datum = new Datum(x,y);  
    }  
    public Datum poll() {  
        Datum d = datum;  
        datum = null;  
        return d;  
    }  
}
```

May lose next datum!

Sample (volatile 2)

```
class Sample {  
    private volatile Datum datum;  
    public void push(int x, int y) {  
        datum = new Datum(x,y);  
    }  
    public Datum poll() {  
        Datum d = datum;  
        if (d.sampled) return null;  
        d.sampled = true;  
        return d;  
    }  
}
```

Sample (volatile 2)

```
class Sample {  
    private volatile Datum datum;  
    public void push(int x, int y) {
```

Only one consumer permitted!

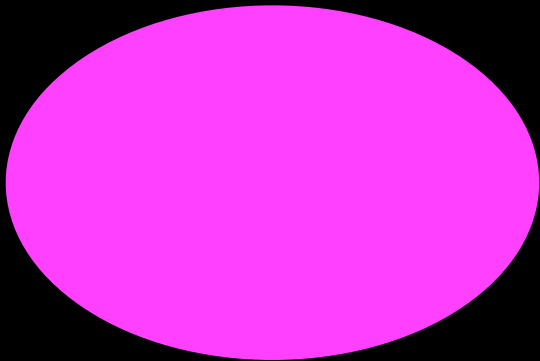
```
    public Datum poll() {  
        Datum d = datum;  
        if (d.sampled) return null;  
        d.sampled = true;  
        return d;  
    }  
}
```

Using Volatile Fields

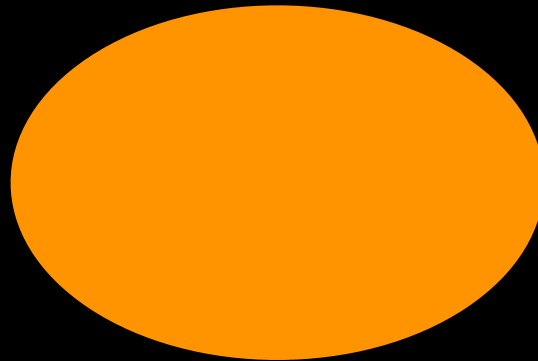
- + Avoids overhead of synchronization;
- + JMM ensures sequential consistency;
- May miss samples
(as with earlier synchronized code too)
- Only one consumer can be permitted in this example.

Transfer (volatile I)

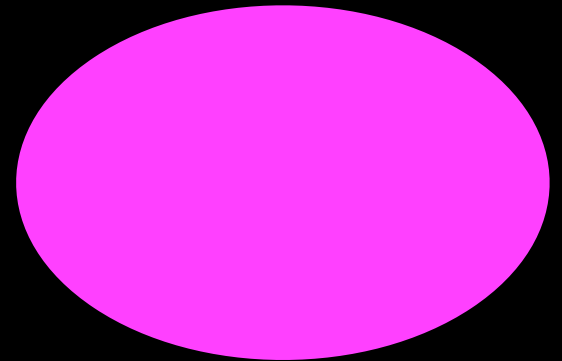
Producer



Sample

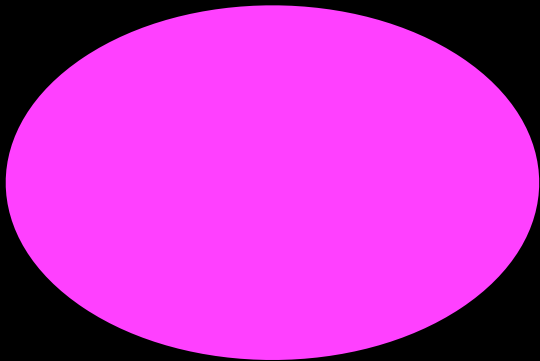


Consumer

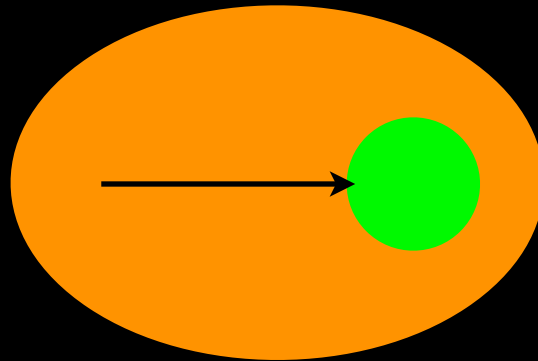


Transfer (volatile I)

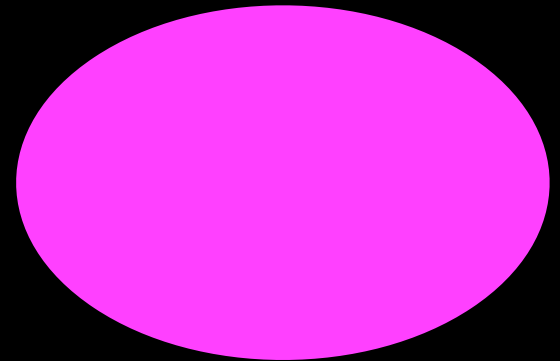
Producer



Sample

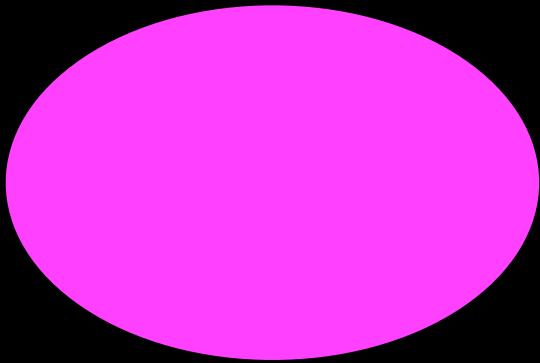


Consumer

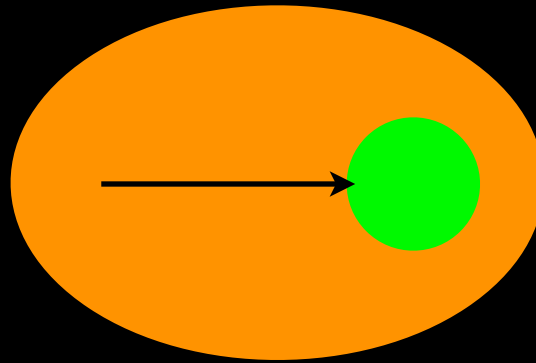


Transfer (volatile I)

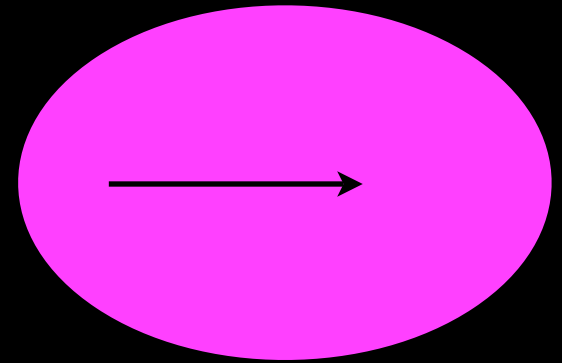
Producer



Sample



Consumer

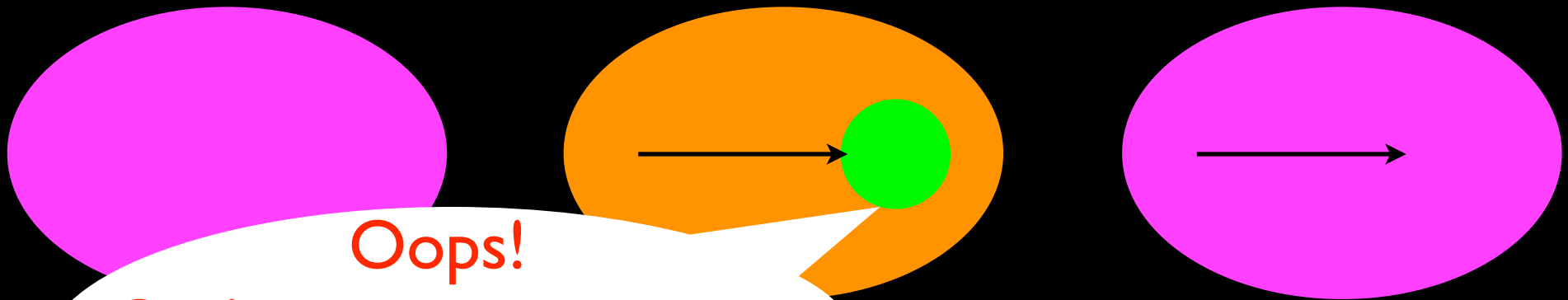


Transfer (volatile I)

Producer

Sample

Consumer

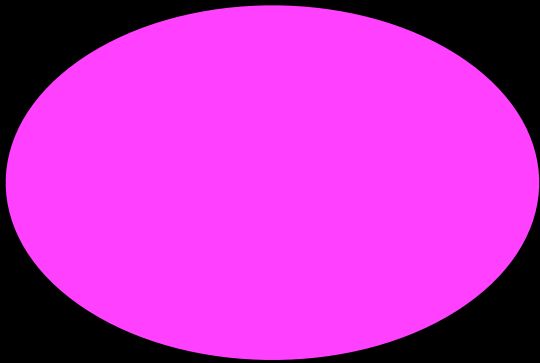


Oops!

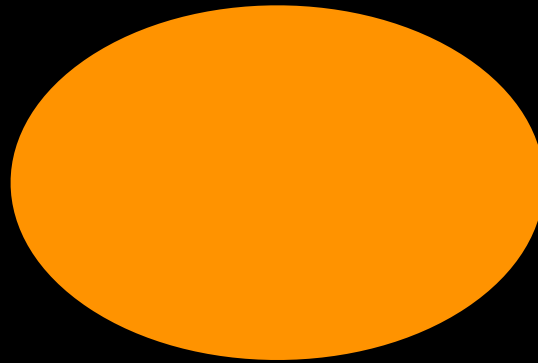
Can't move permission
without synchronization

Transfer (volatile 2)

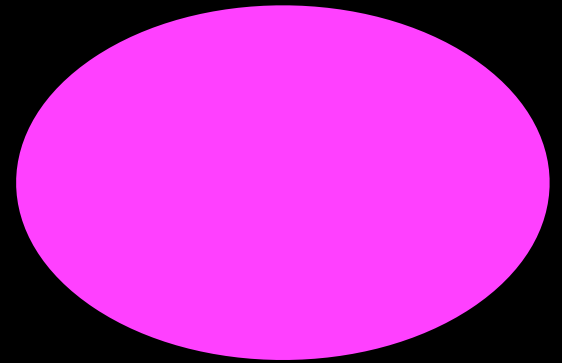
Producer



Sample

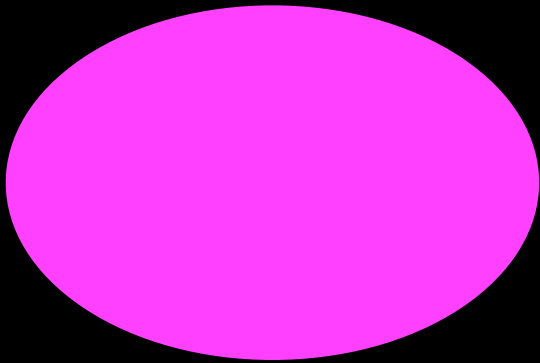


Consumer

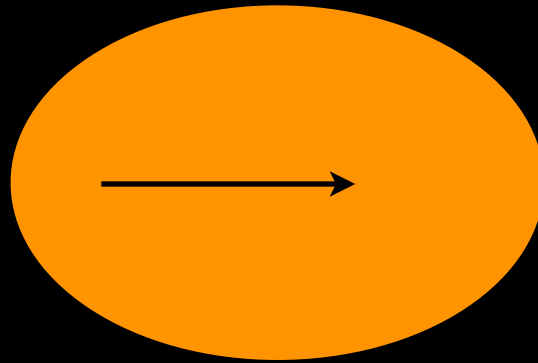


Transfer (volatile 2)

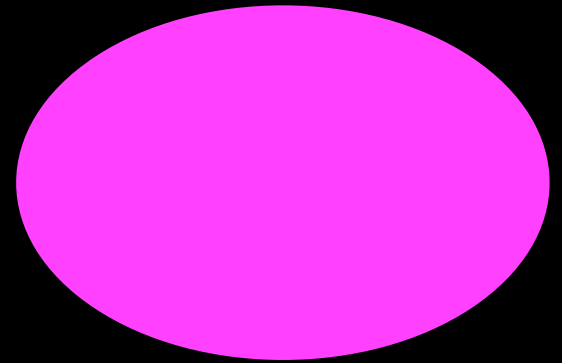
Producer



Sample

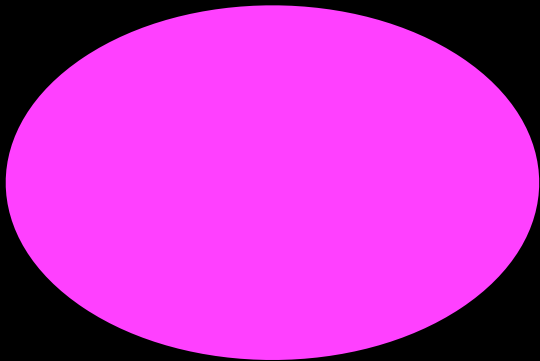


Consumer

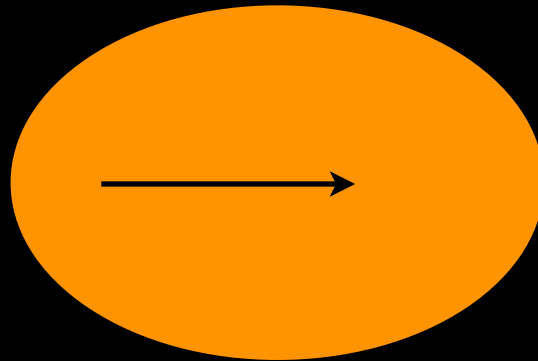


Transfer (volatile 2)

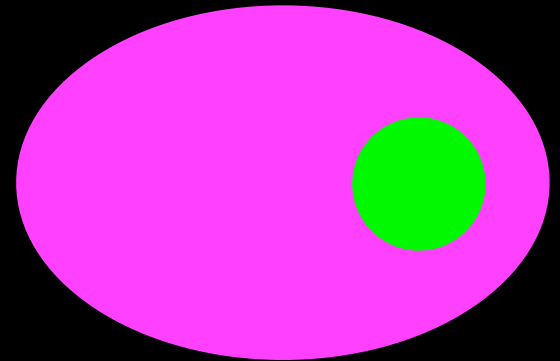
Producer



Sample

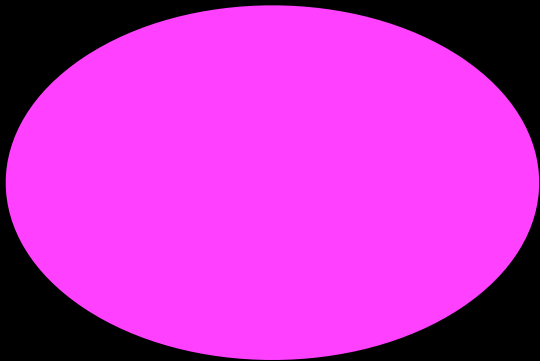


Consumer

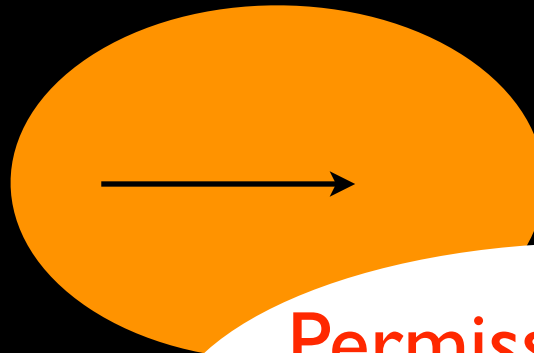


Transfer (volatile 2)

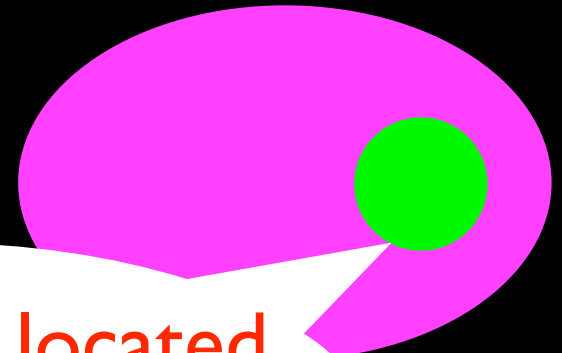
Producer



Sample

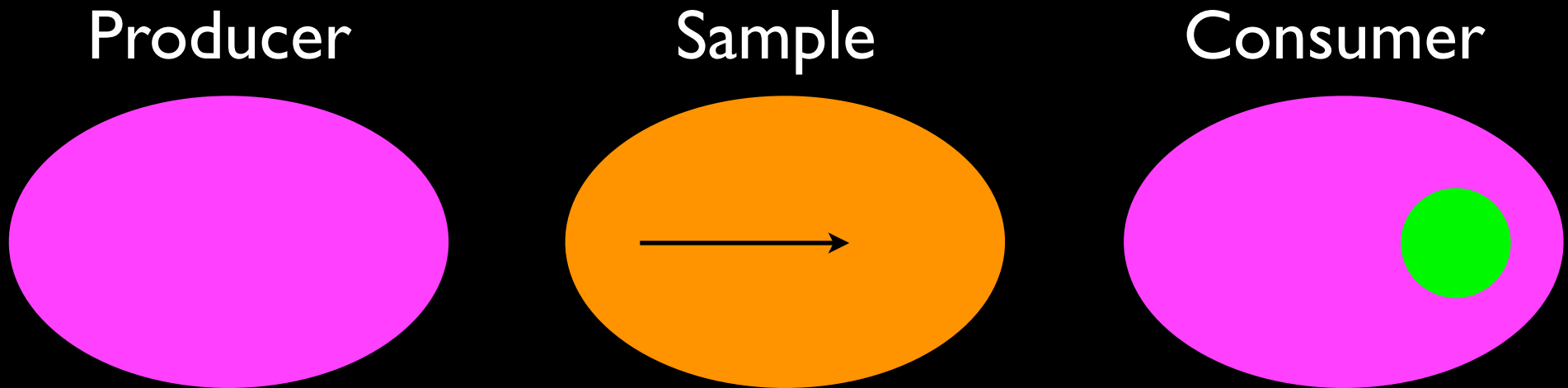


Consumer



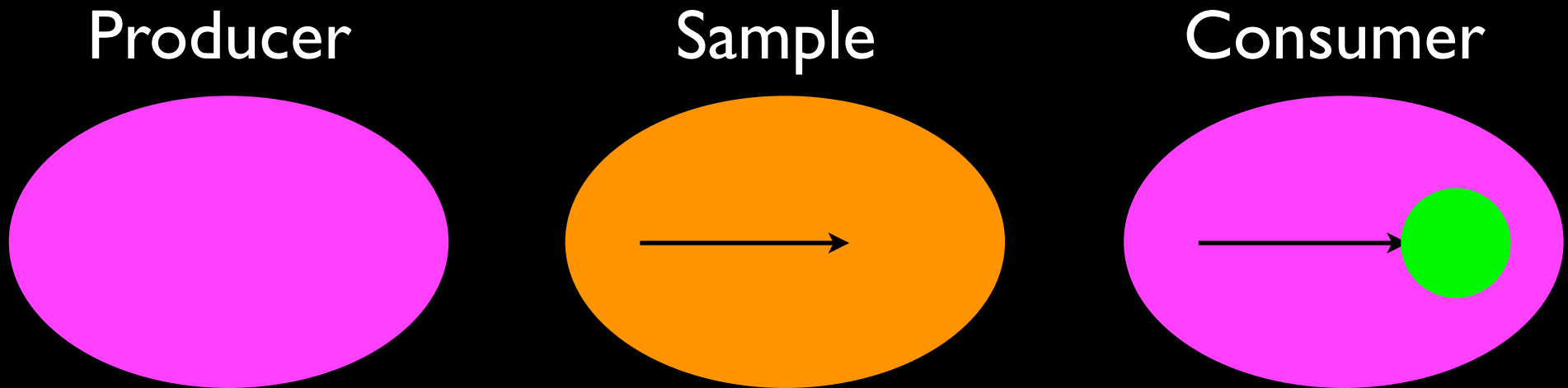
Permission located
"somewhere"
in consumer

Transfer (volatile 2)



Sample Invariant:
unless null, the permission is located in consumer.

Transfer (volatile 2)



Sample Invariant:
unless null, the permission is located in consumer.

“P Located in X”

- a NONLINEAR fact;
- implied by (Clarke-style) ownership;
- formalized as “adoption” [FDL02] and “nesting” [BR05];
- ownership systems handle volatile easily [Clarke, Müller, Boyapati, Greenhouse]

Linear Strikes Back!

- Linear systems can handle volatile too:

```
atomic {  
    d = datum;  
    if (state == NEW) ...  
    state = READ;  
}
```

- synchronization for **model** data only.

Conclusions

1. Volatile fields may be read zero, once or many times;
2. A nonlinear invariant is thus easier to use;
3. Ownership (adoption) is a prime example;
4. Linear reasoning is still possible through introduction of atomic blocks manipulating auxiliary data.

