Java RaceFinder

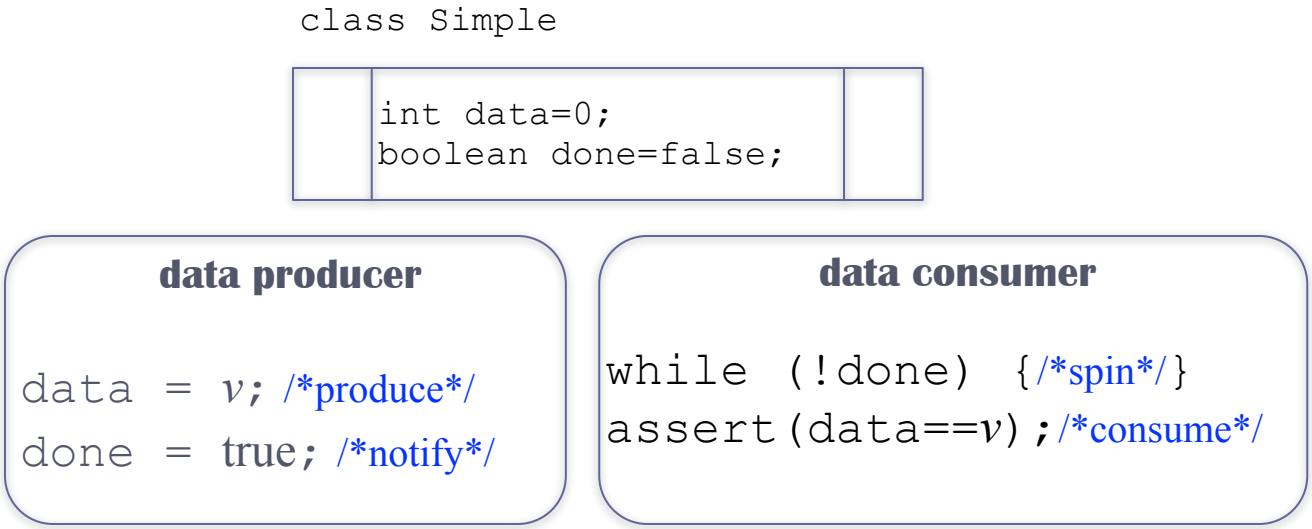
Precise Data Race Detector
for a Relaxed Memory Model

By KyungHee Kim
Department of Computer &
Information Science & Engineering
University of Florida

Motivating Example

Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



Motivating Example

- Simple java program with two threads **data producer** and **data consumer**
- The property is "(done=true) implies (data=*v*)".

```
class Simple

    int data=0;
    boolean done=false;
```

data producer

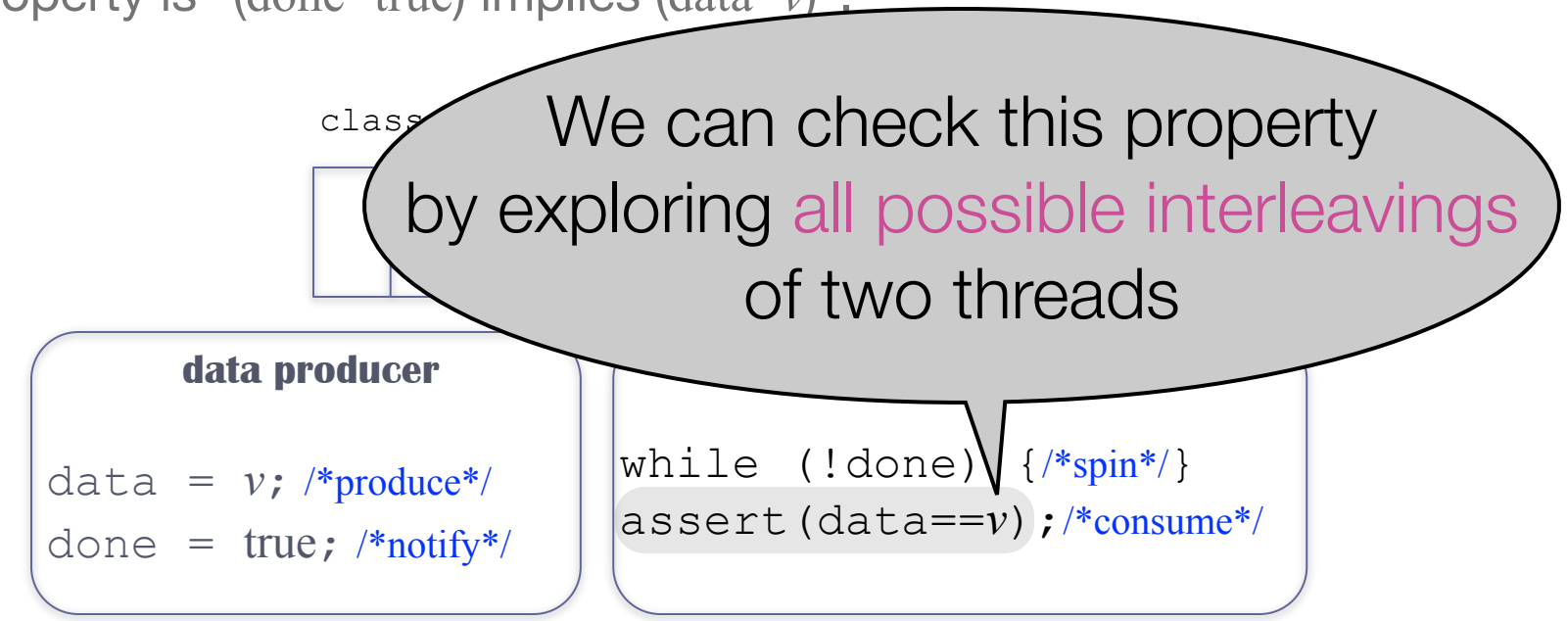
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

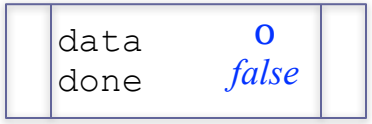
```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

Motivating Example

- Simple java program with two threads **data producer** and **data consumer**
- The property is "(done=true) implies (data=*v*)".



Motivating Example

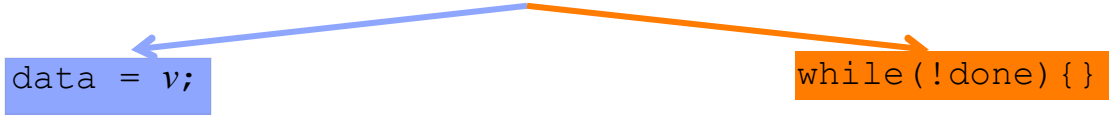


data producer

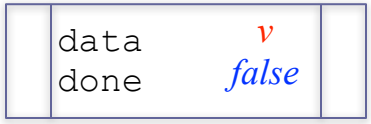
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



Motivating Example

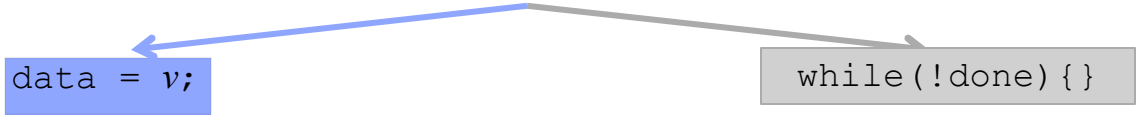


data producer

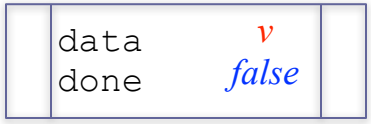
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



Motivating Example

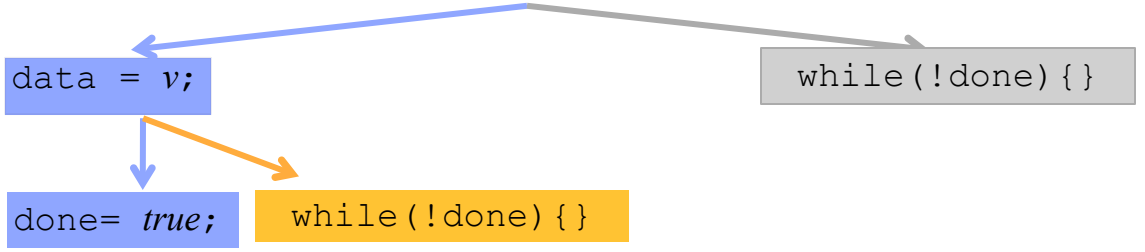


data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



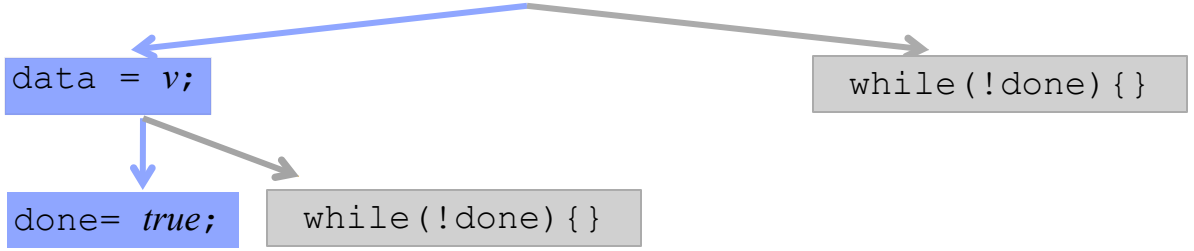
Motivating Example

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



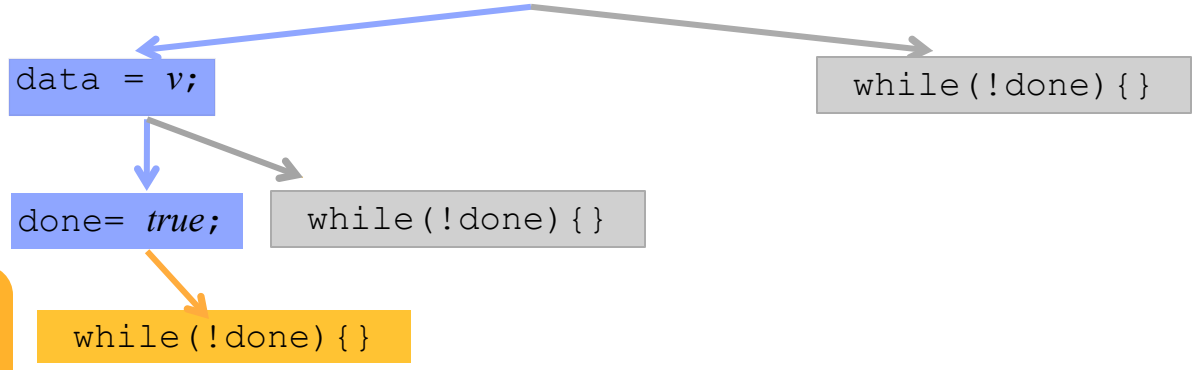
Motivating Example

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



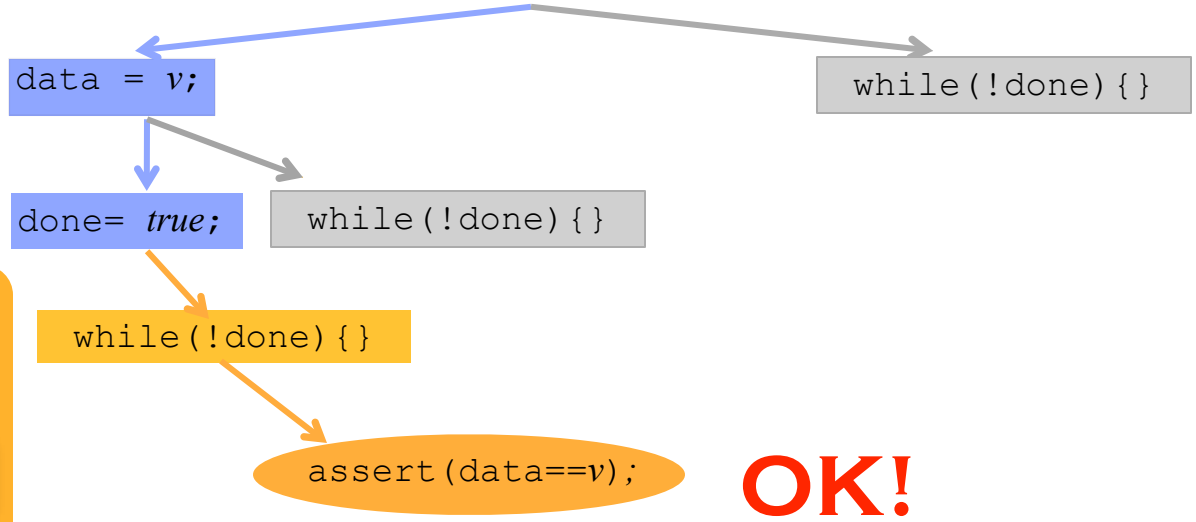
Motivating Example

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



Motivating Example

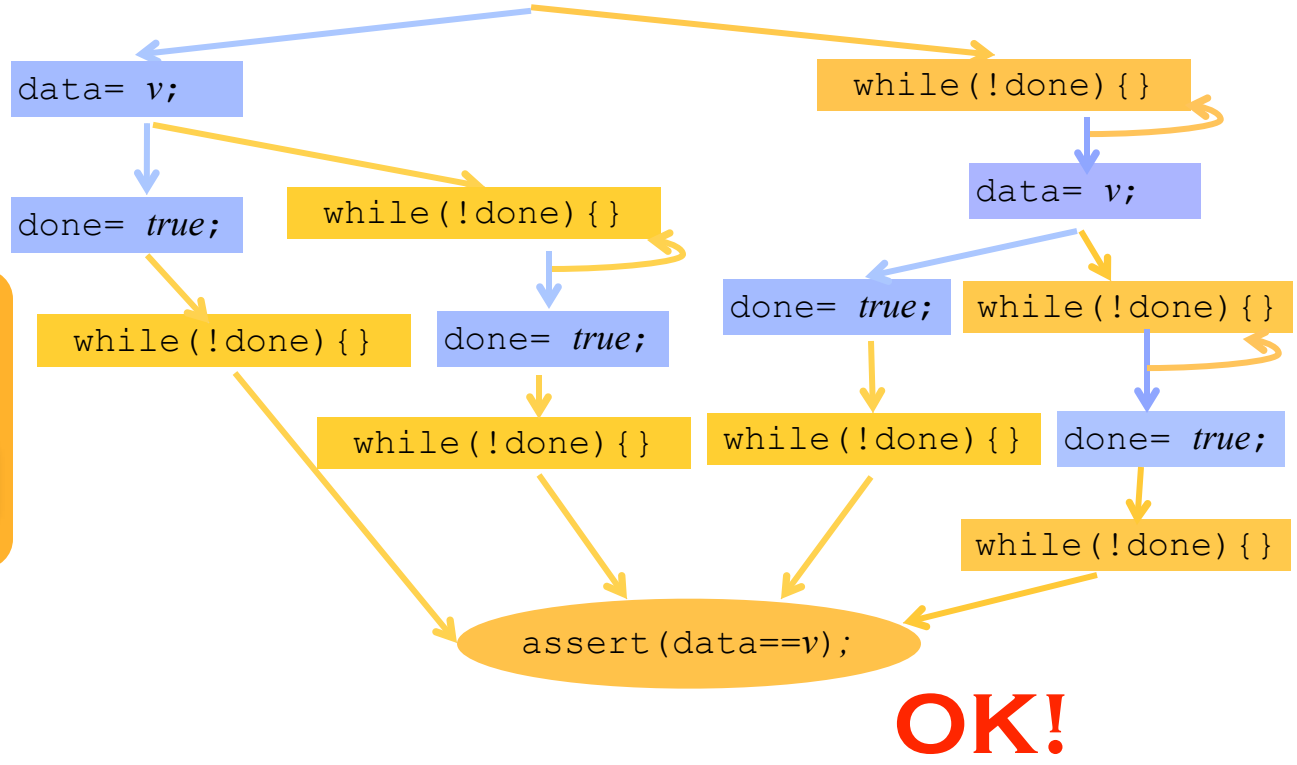
data	*v*
done	*true*

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



Motivating Example

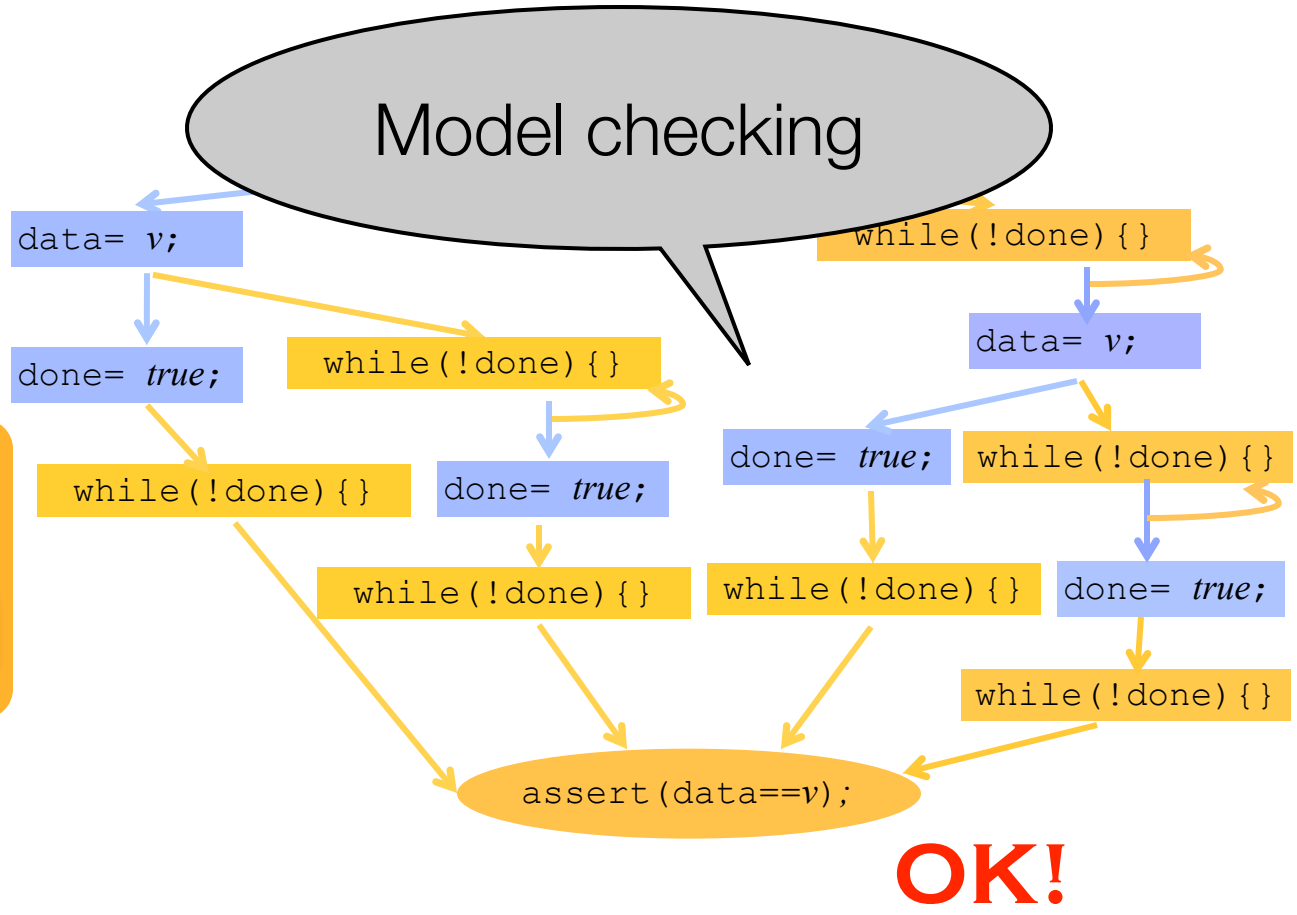
data	*v*
done	*true*

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```



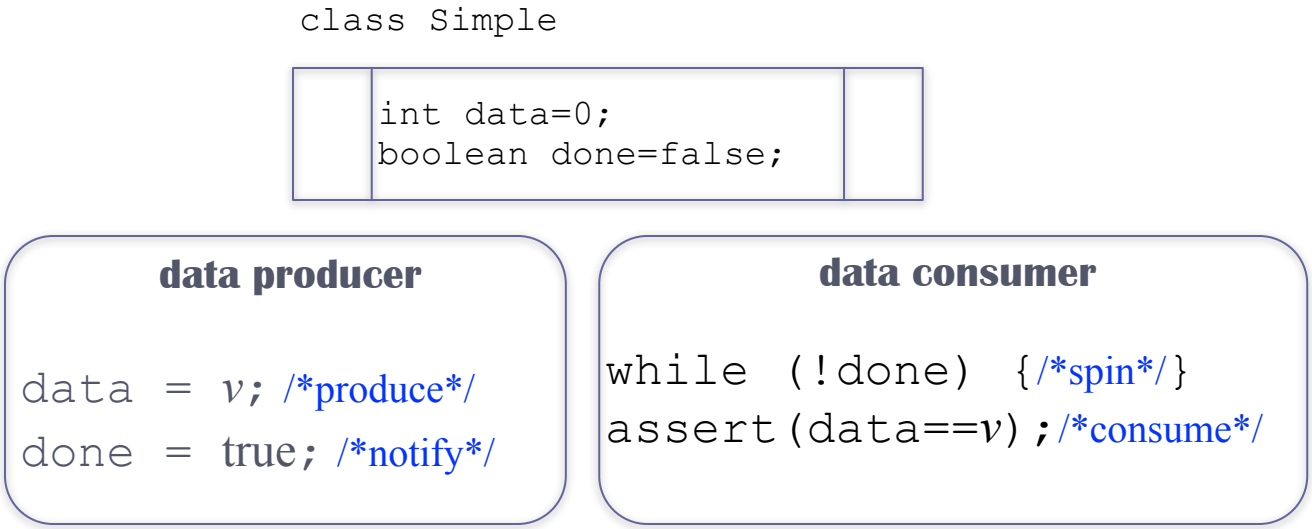
Motivating Example

- Simple java program with two threads **data producer** and **data consumer**
- The property is "(done=true) implies (data=*v*)"


```
JavavPathFinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center

====================================================== system under test
application: simple/Simple.java

====================================================== search started: 2/16/10 2:18 AM

====================================================== results
no errors detected
====================================================== statistics
elapsed time:       0:00:00
states:           new=7, visited=4, backtracked=10, end=2
search:            maxDepth=3, constraints=0
choice generators:  thread=7, data=0
heap:             gc=6, new=283, free=30
instructions:       2925
max memory:         80MB
loaded code:        classes=70, methods=982

====================================================== search finished: 2/16/10 2:18 AM
```

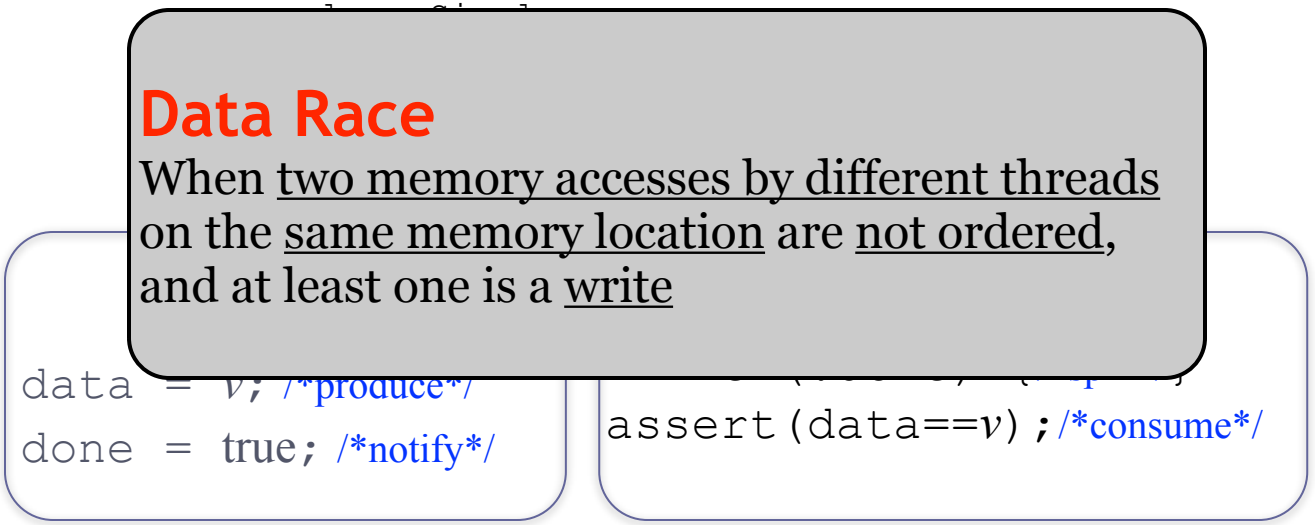
Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



Motivating Example

- Simple java program with two threads **data producer** and **data consumer**

```
class Simple

    int data=0;
    boolean done=false;
```

data producer

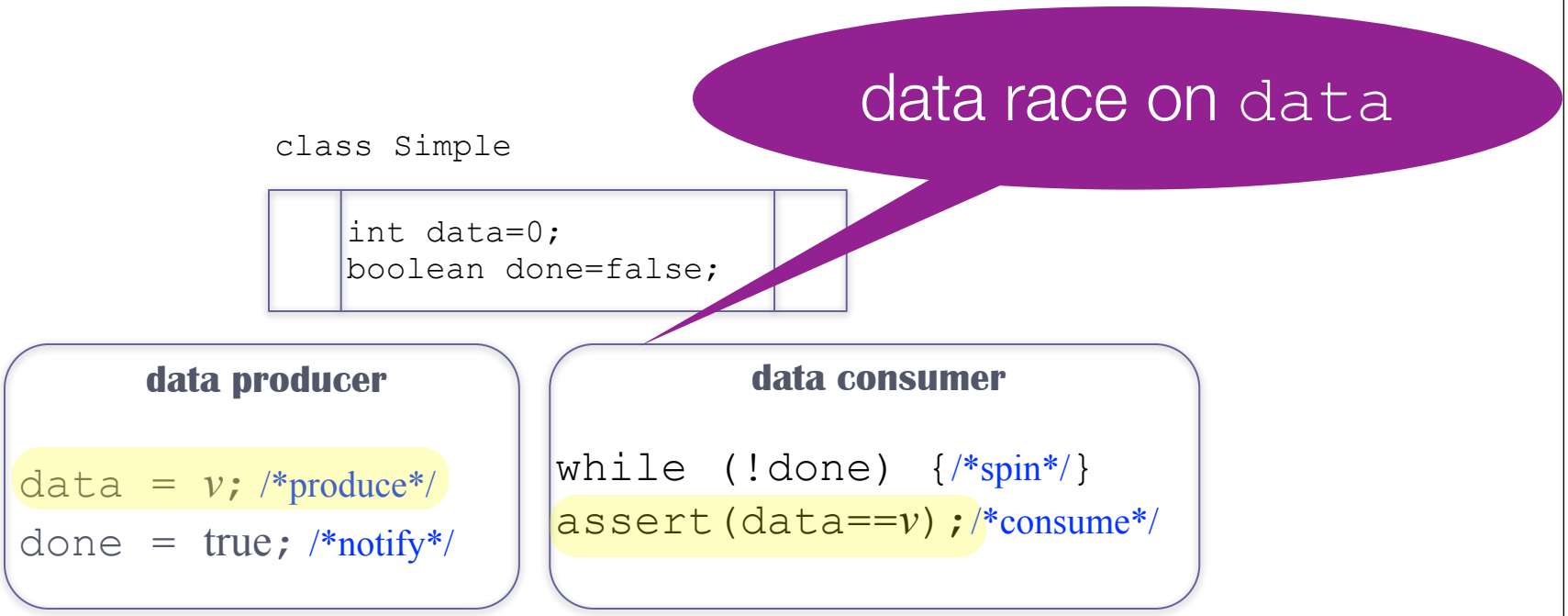
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

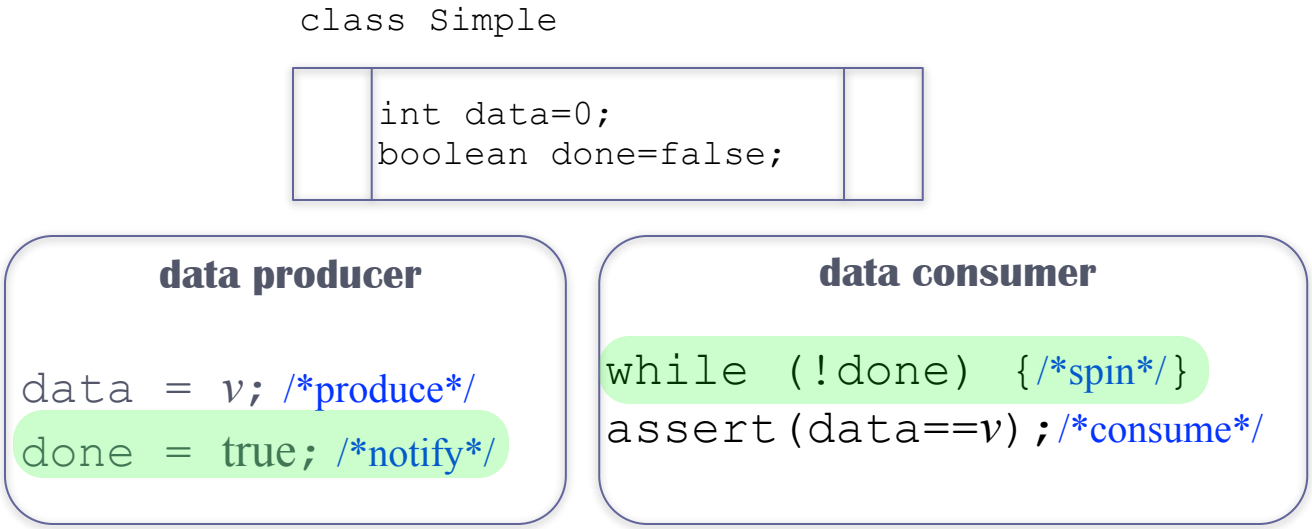
Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



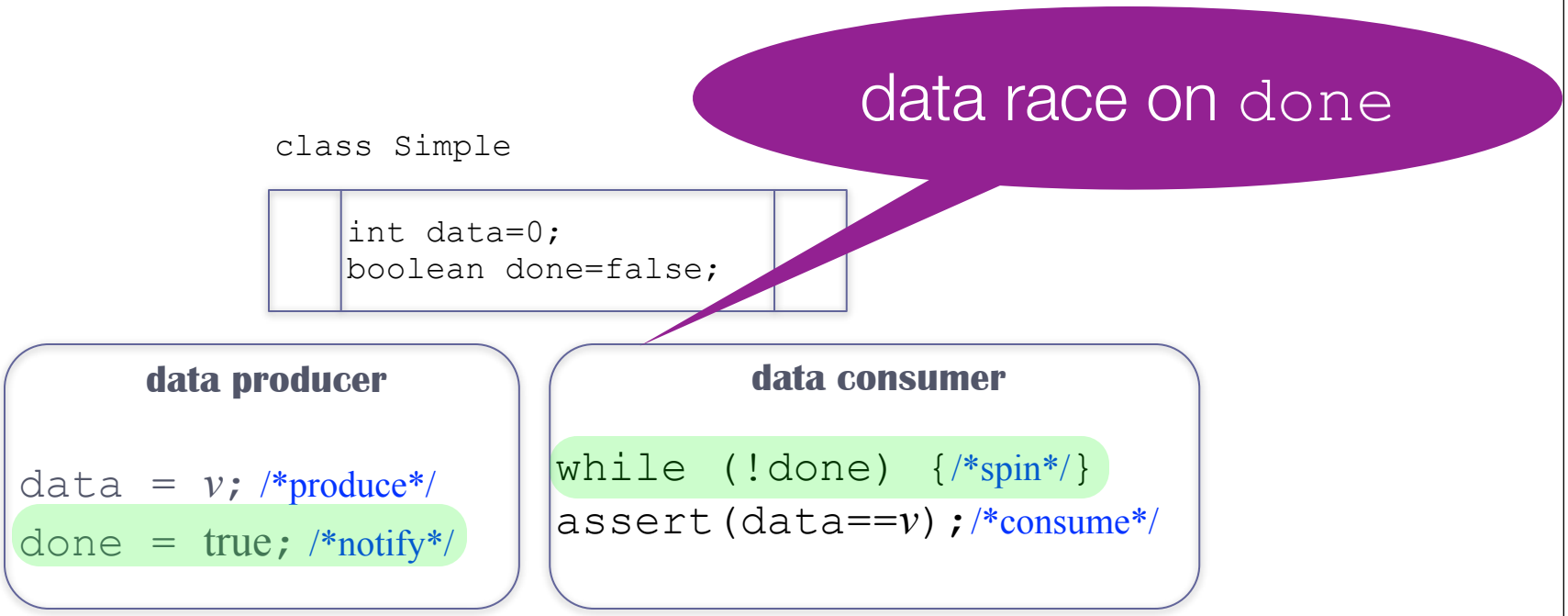
Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



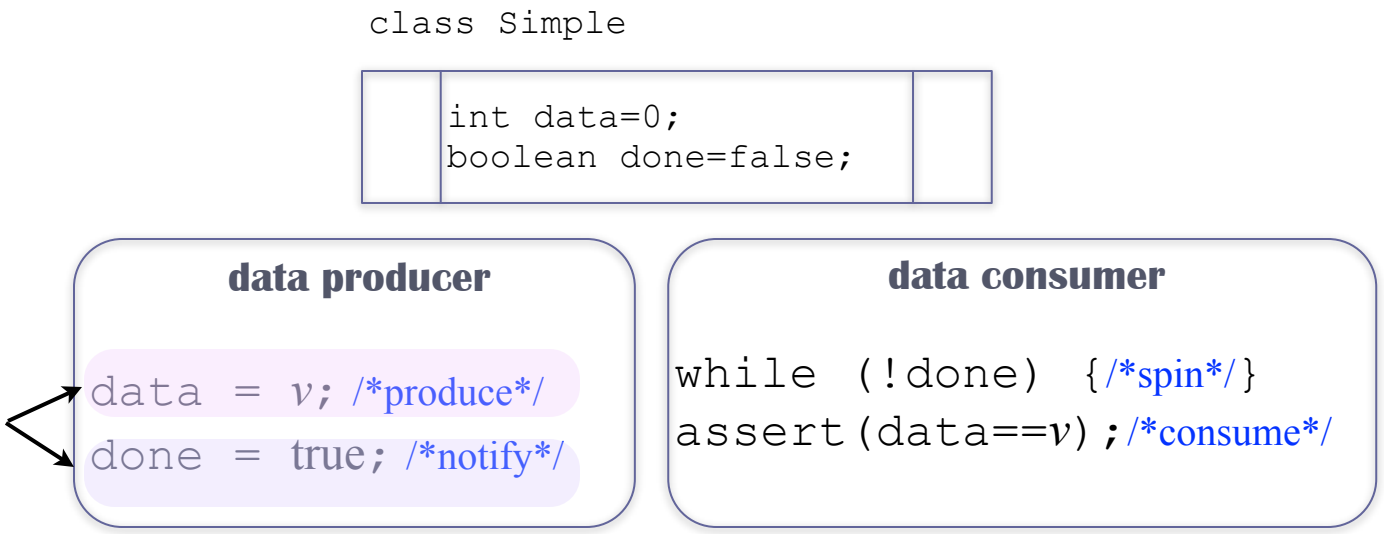
Motivating Example

- Simple java program with two threads **data producer** and **data consumer**

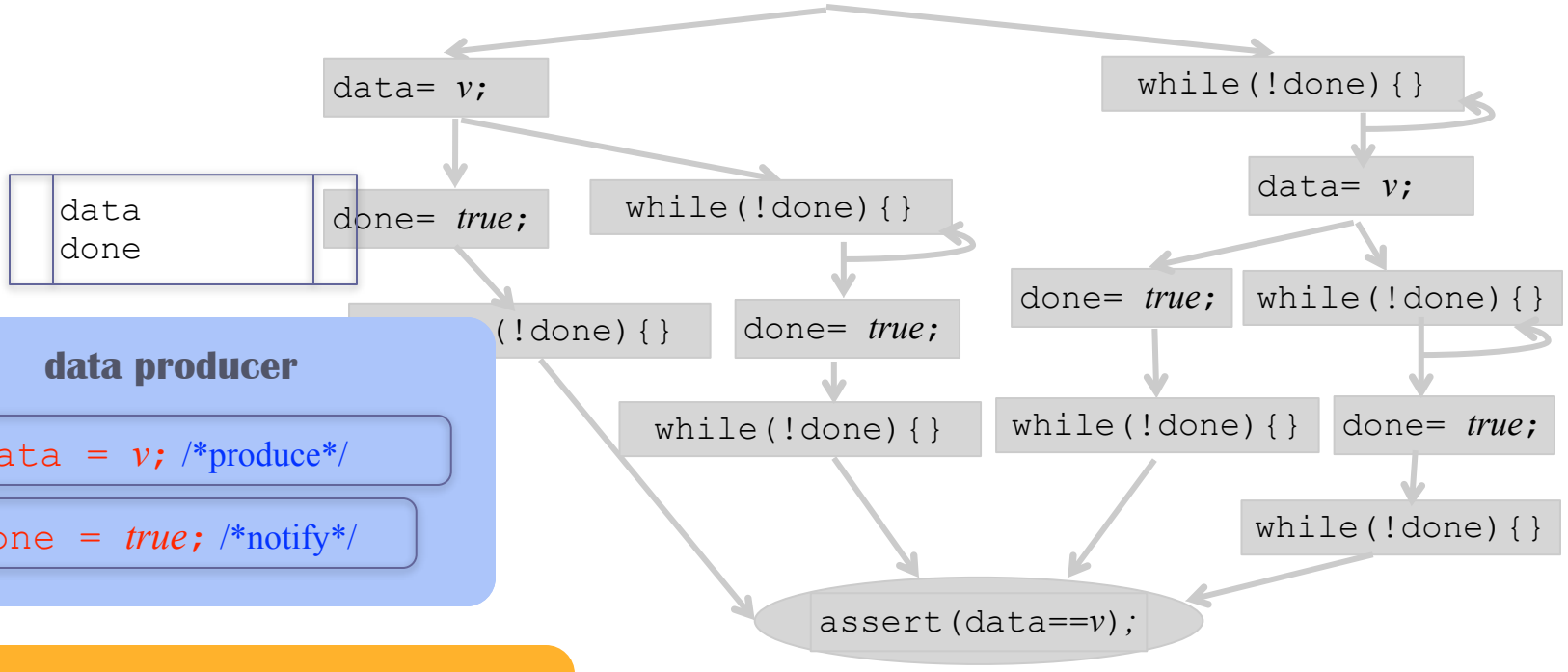


Motivating Example

- Simple java program with two threads **data producer** and **data consumer**



Motivating Example



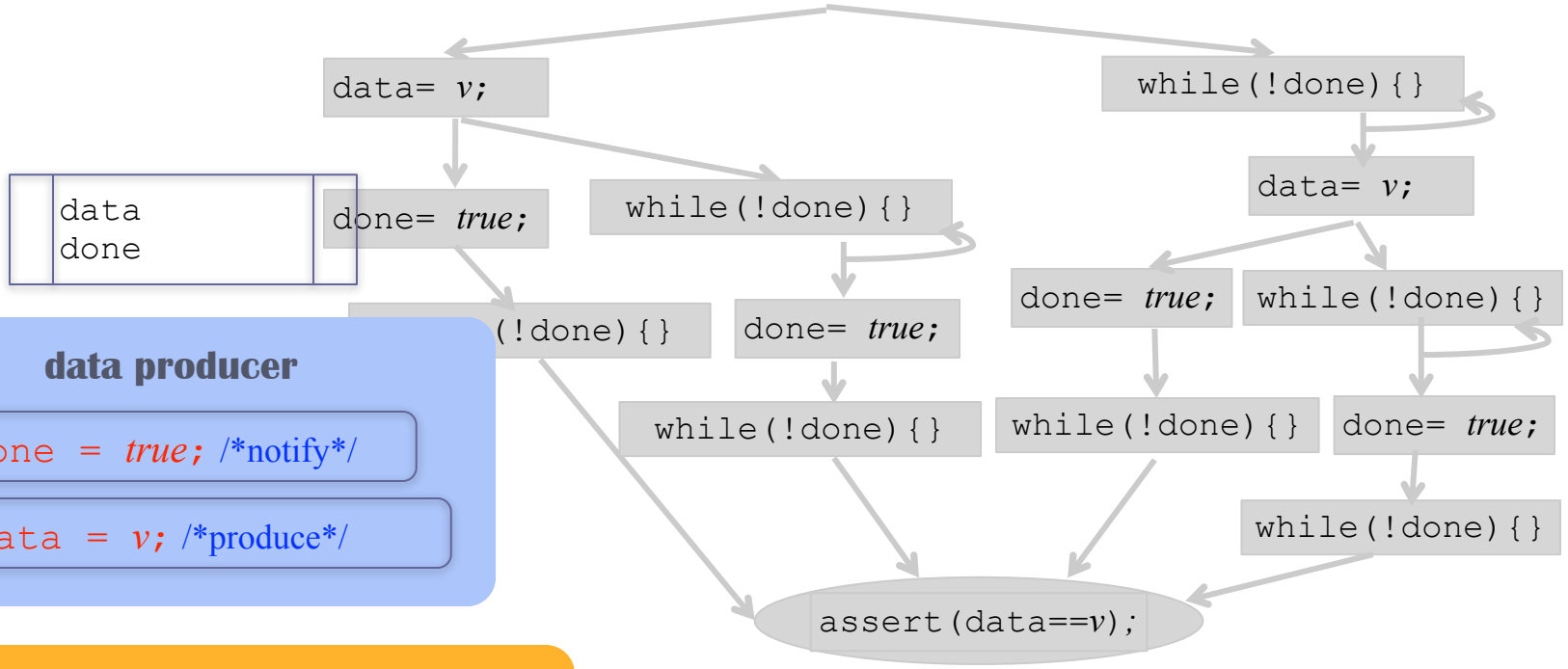
data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```

Motivating Example



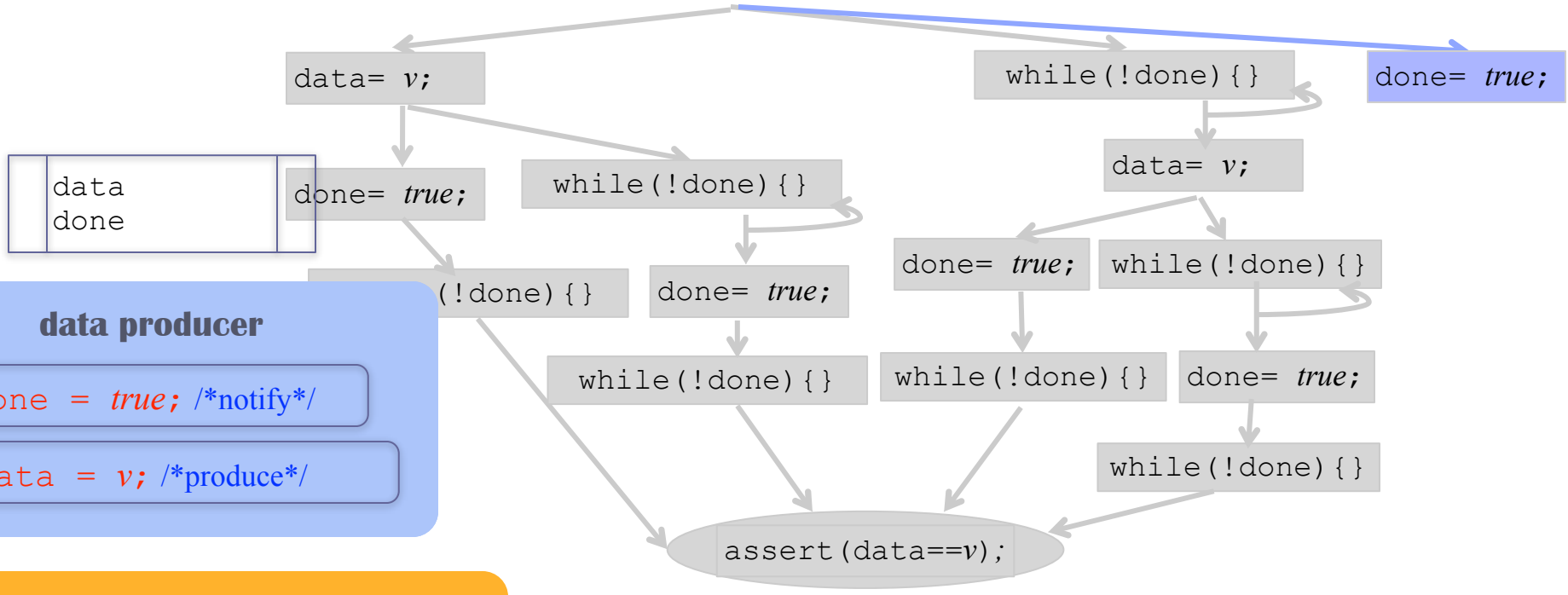
data producer

```
done = true; /*notify*/
data = v; /*produce*/
```

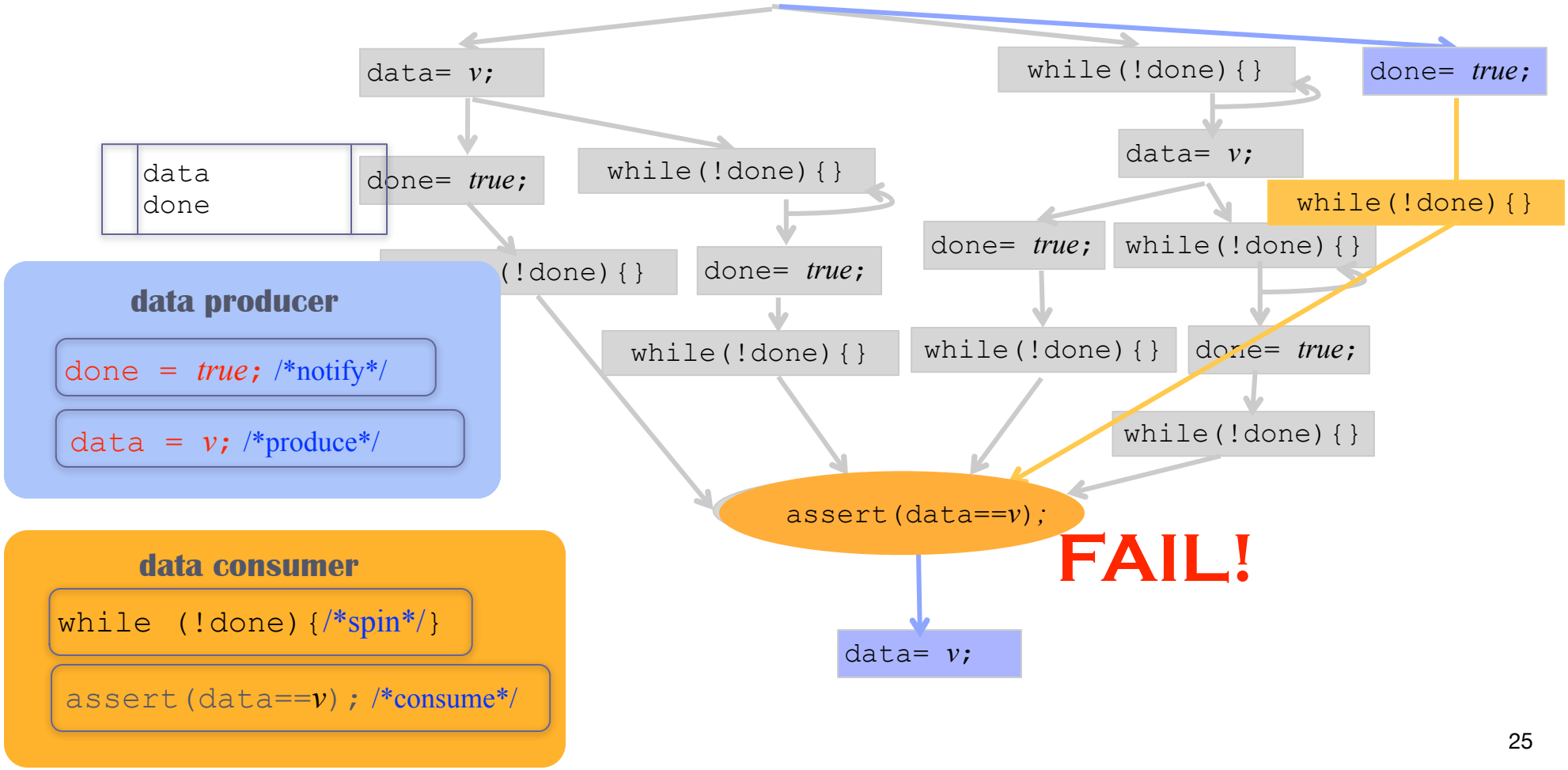
data consumer

```
while (!done){/*spin*/}
assert(data==v); /*consume*/
```

Motivating Example



Motivating Example



Motivating Example

- How to correct the program?

two memory accesses
by different threads
✓ no ordering
at least one is write

- add a memory barrier to order the memory accesses

Motivating Example

(1) use a synchronization

```
class SimpleSync

    int data=0;
    boolean done=false;
```

data producer

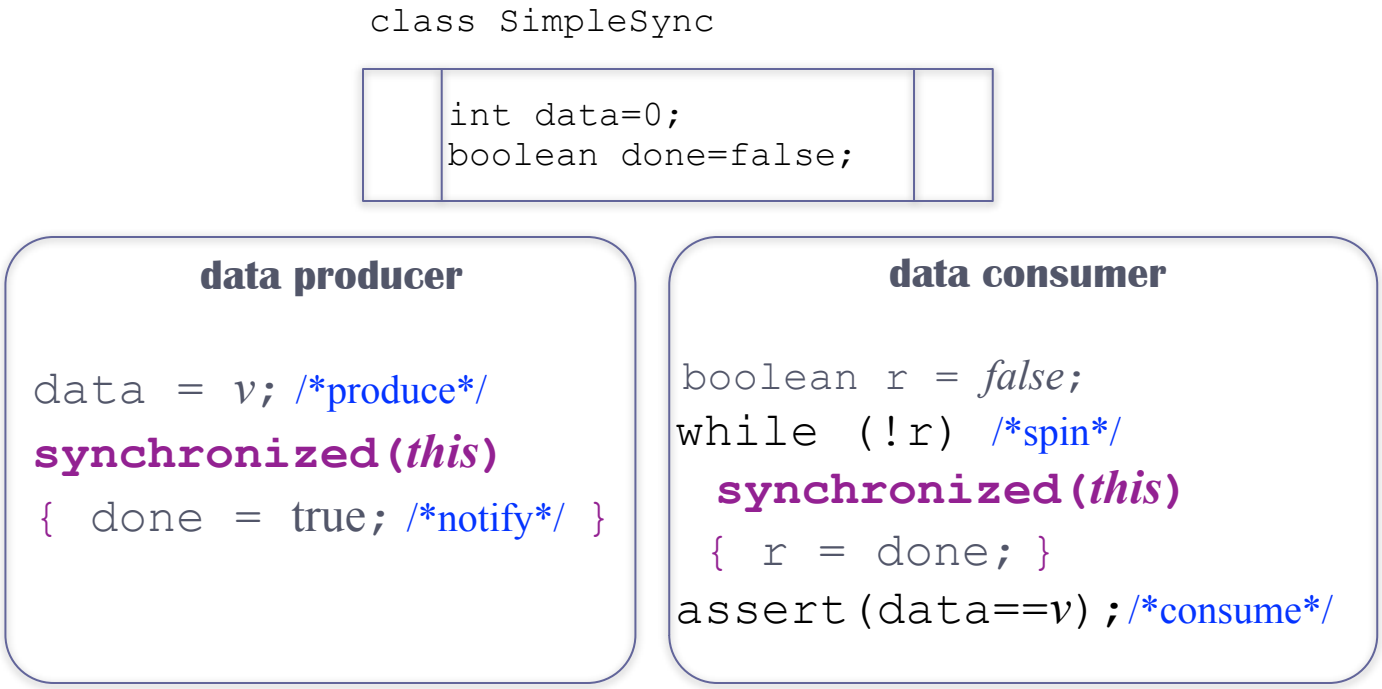
```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done){} /*spin*/
assert(data==v);/*consume*/
```

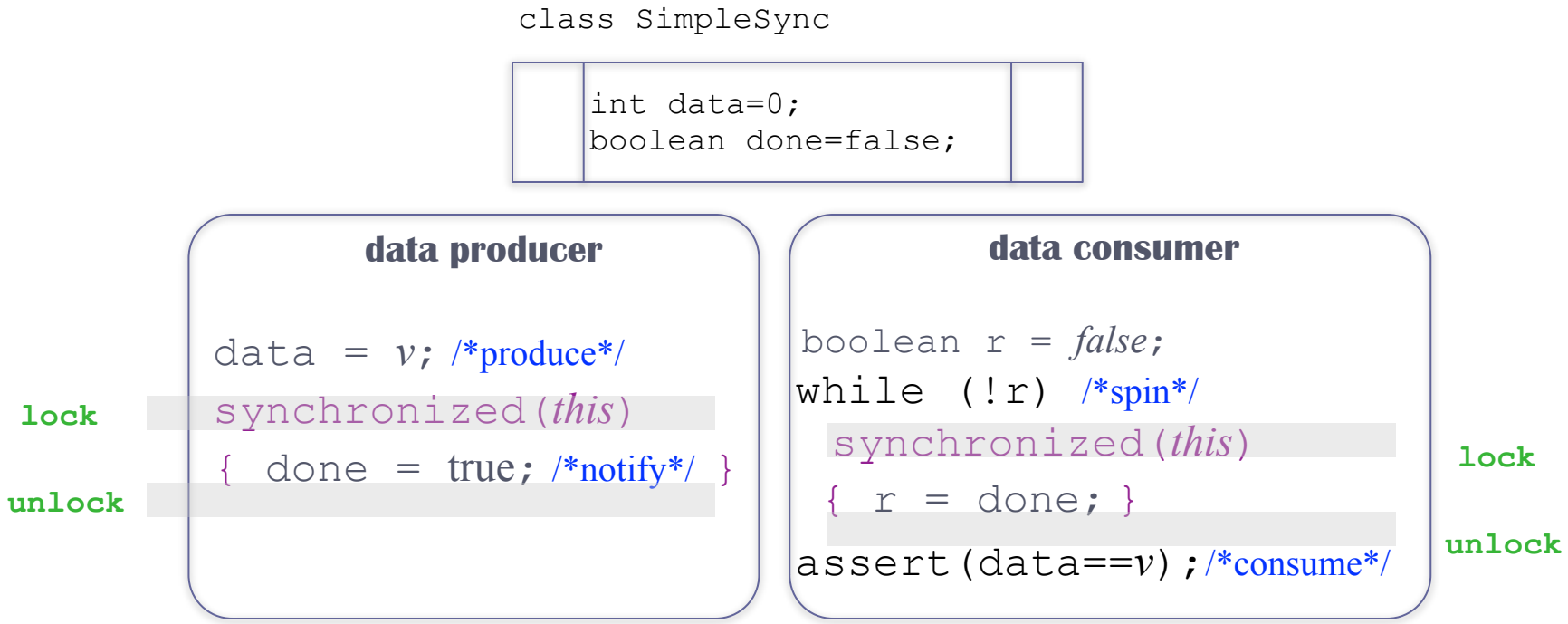
Motivating Example

(1) use a synchronization



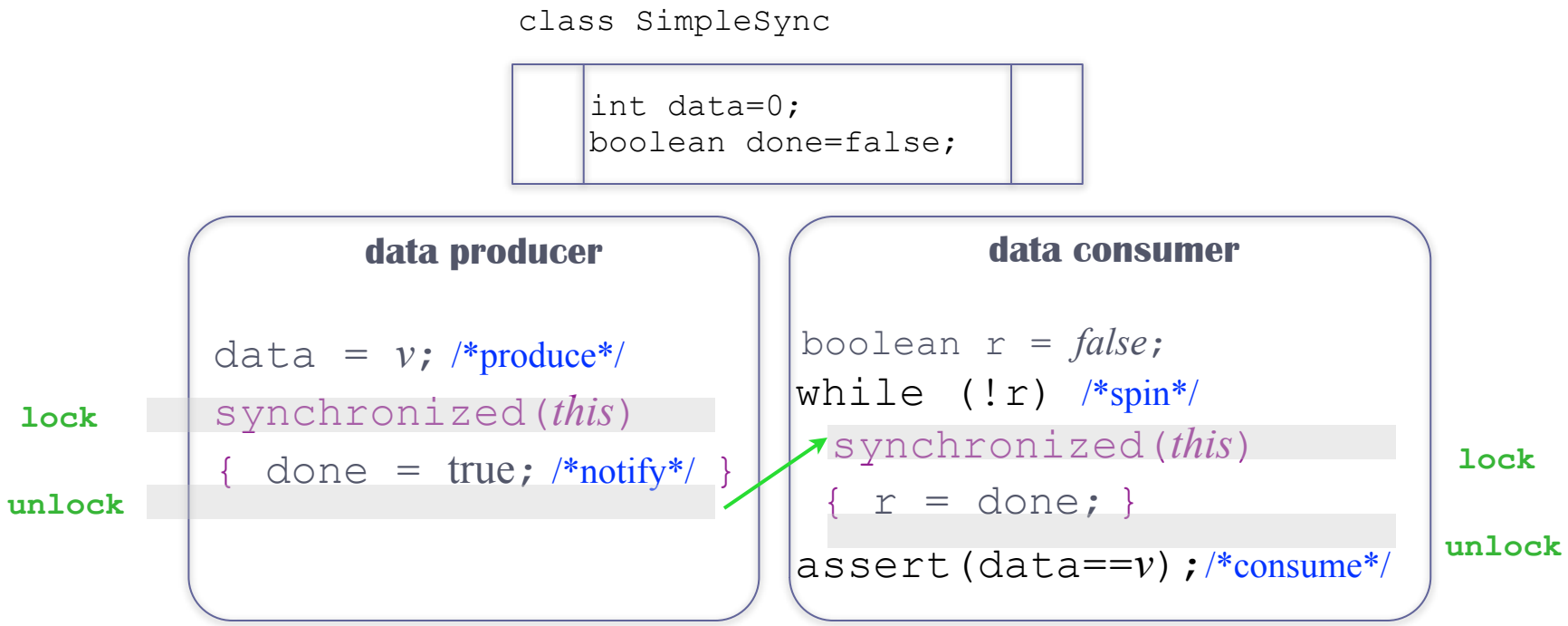
Motivating Example

(1) use a synchronization



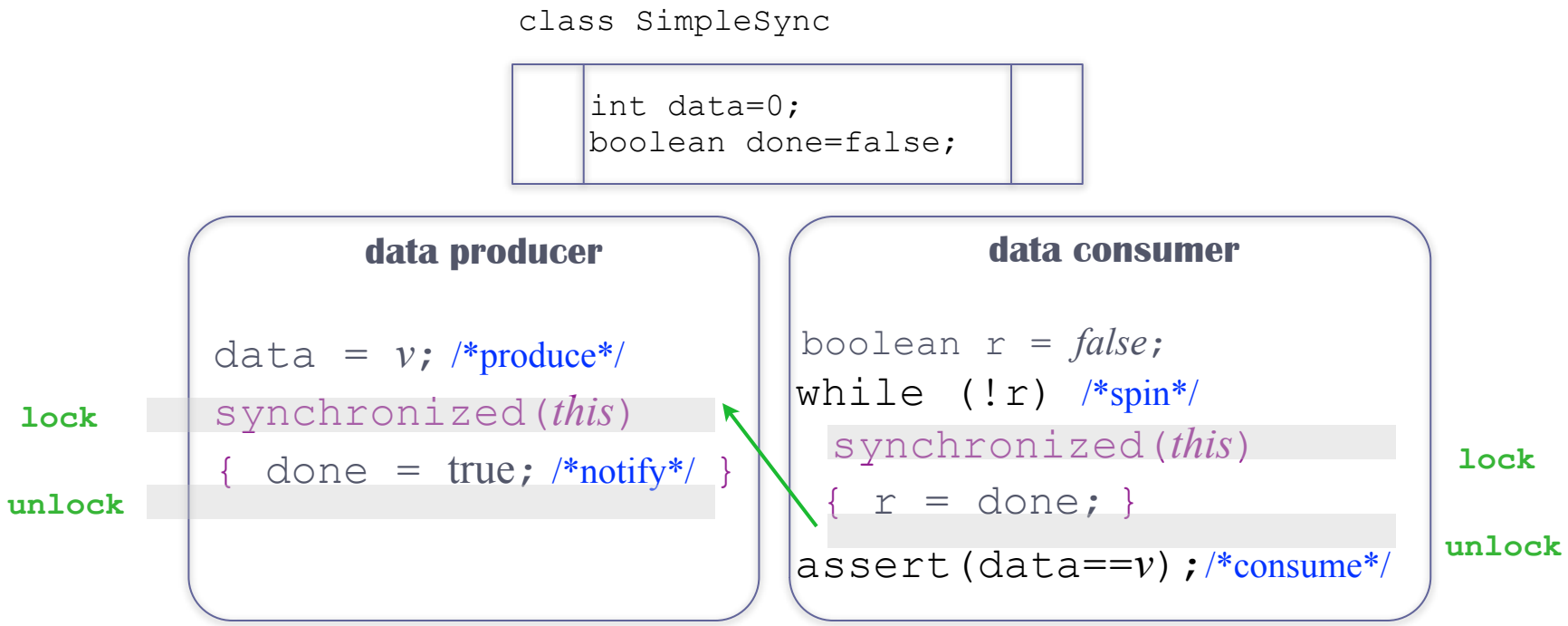
Motivating Example

(1) use a synchronization



Motivating Example

(1) use a synchronization



Motivating Example

(2) use a memory barrier using "`volatile`" keyword

```
class SimpleVolatile
```

```
int data=0;
boolean done=false;
```

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

Motivating Example

(2) use a memory barrier using "`volatile`" keyword

```
class SimpleVolatile

    int data=0;
    volatile boolean done=false;
```

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

Motivating Example

(2) use a memory barrier using "`volatile`" keyword

```
class SimpleVolatile
```

```
int data=0;
volatile boolean done=false;
```

data producer

```
data = v; /*produce*/
done = true; /*notify*/
```

data consumer

```
while (!done) {/*spin*/}
assert(data==v);/*consume*/
```

What is “Java RaceFinder”?

What is “Java RaceFinder”?

- Precise Data Race Detector for a Relaxed Memory Model
- Target environment is Java
- The goal is to find a race

What is “Java RaceFinder”?

- Precise Data Race Detector in a Relaxed Memory Model
- Target environment is Java
- The goal is to find a data race

  => to verify the correctness of a concurrent program,

  JRF detects a data race in a multithreaded program

What is “Java RaceFinder”?

- Precise Data Race Detector in a Relaxed Memory Model
- Target environment is Java
- The goal is to find a data race

  => to verify the correctness of a concurrent program,

  JRF detects a data race in a multithreaded program

  - two memory accesses
  - different threads
  - no ordering
  - at least one is write

Java Memory Model

Java Memory Model

Memory Model
defines exactly how threads interact with memory and how the programmer can control

Java Memory Model

Memory Model
defines exactly how threads interact with memory and how the programmer can control

Programmer's view:
which write can be seen by read

Java Memory Model

Memory Model

defines exactly how threads interact with memory and how the programmer can control

Sequential Consistency

A concurrent program behaves as if
- all of its atomic actions occur in some global order
  (consistent with the program order on each thread)
- all threads see values written to main memory in a consistent order

Java Memory Model

Memory Model
defines exactly how threads interact with memory and how the programmer can control

Sequential Consistency

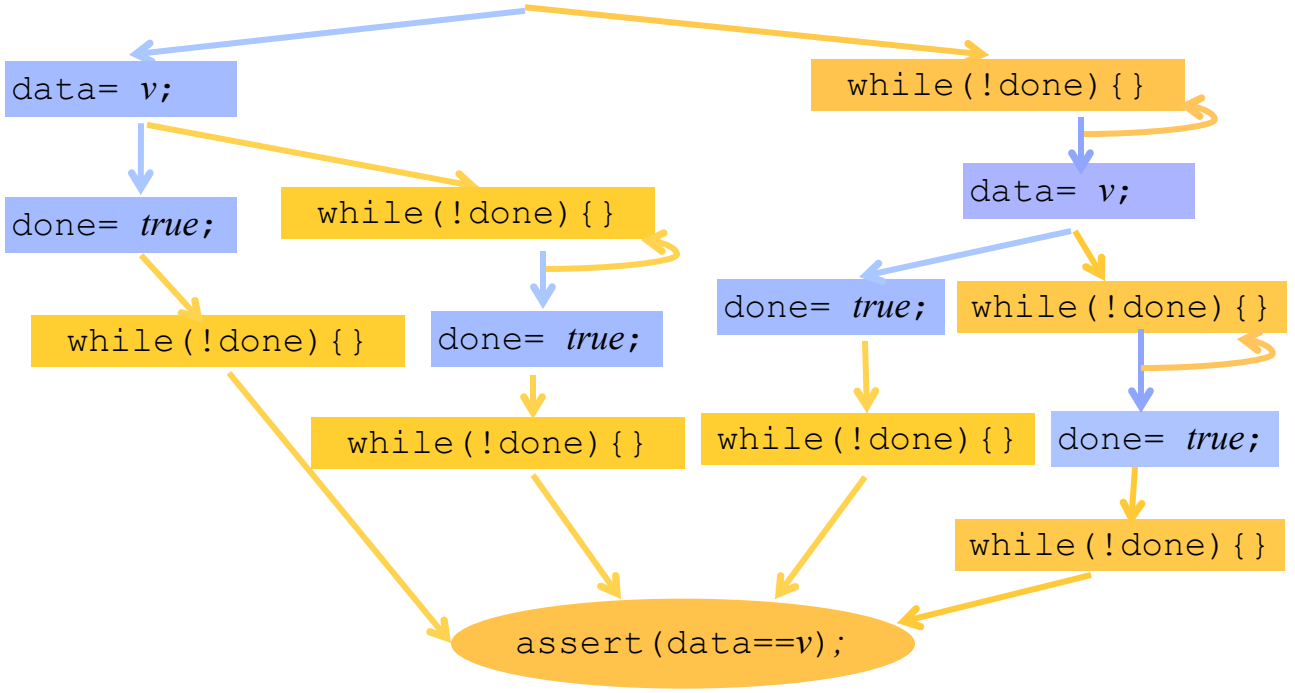
A concurrent program behaves as if
- all of its atomic actions occur in some global order
  (consistent with the program order on each thread)
- all threads see values written to main memory in a consistent order

Relaxed Memory Model

- Sequential consistency is not guaranteed.
- threads might not see values written to main memory in a consistent order

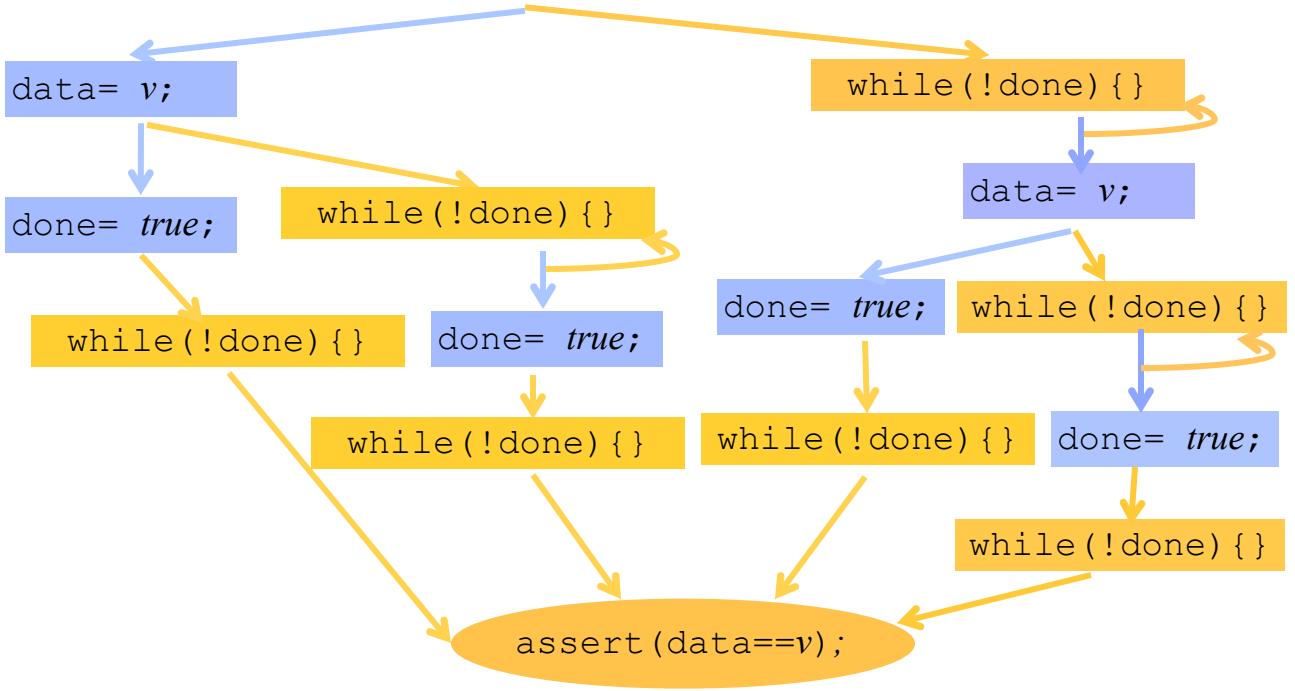
Java Memory Model

- Sequentially Consistent Executions of `Simple` class



Java Memory Model

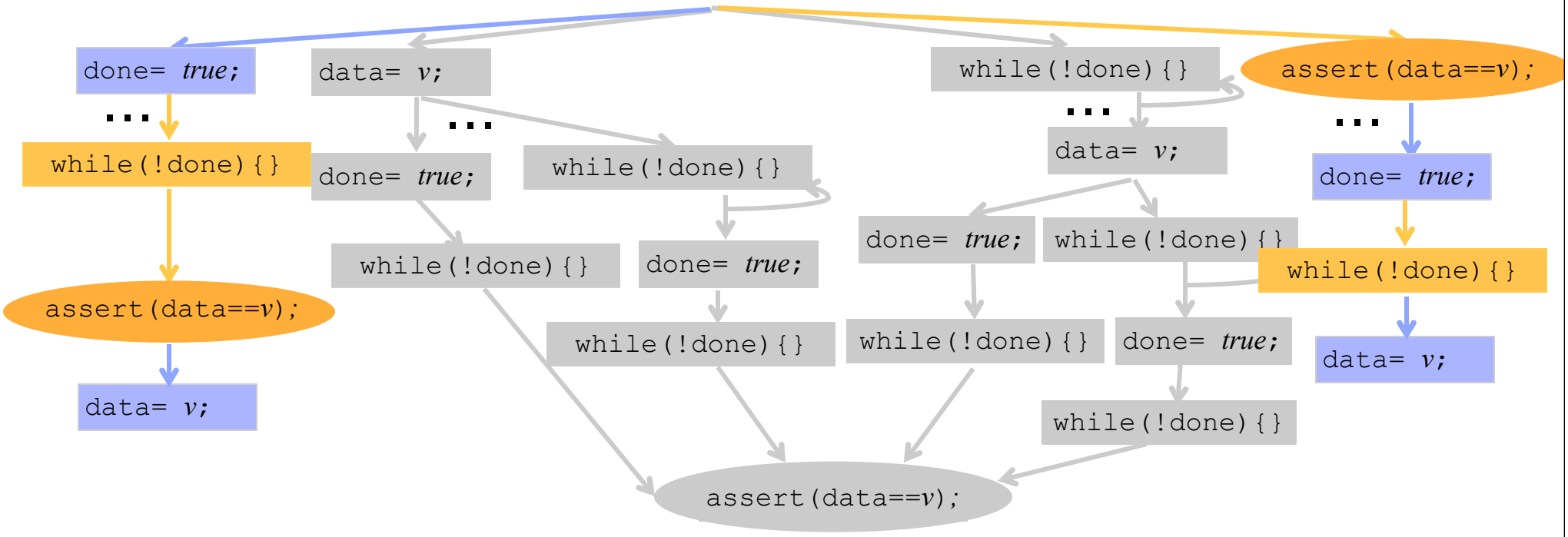
- Sequentially Consistent Executions of `Simple` class



for both **data producer** and **data consumer**, `data=v` then `done=true`

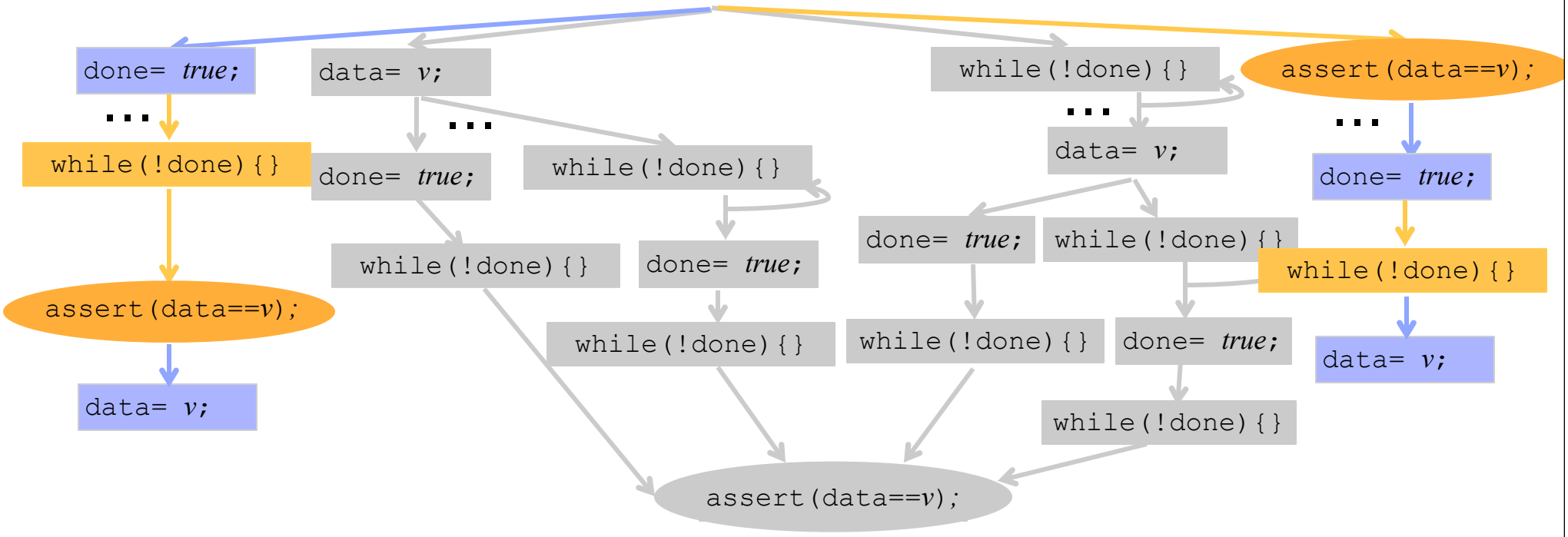
Java Memory Model

- Possible Non-Sequentially Consistent Execution of `Simple` class



Java Memory Model

- Possible Non-Sequentially Consistent Execution of `Simple` class



for **data producer**, `data=`*v* then `done=`*true*
but for **data producer**, `done=`*true* but `data`≠*v*

Java Memory Model

- a relaxed memory model.
- a data race condition is,

  two memory accesses
  by different threads
  not ordered by happens-before order
  at least one is write

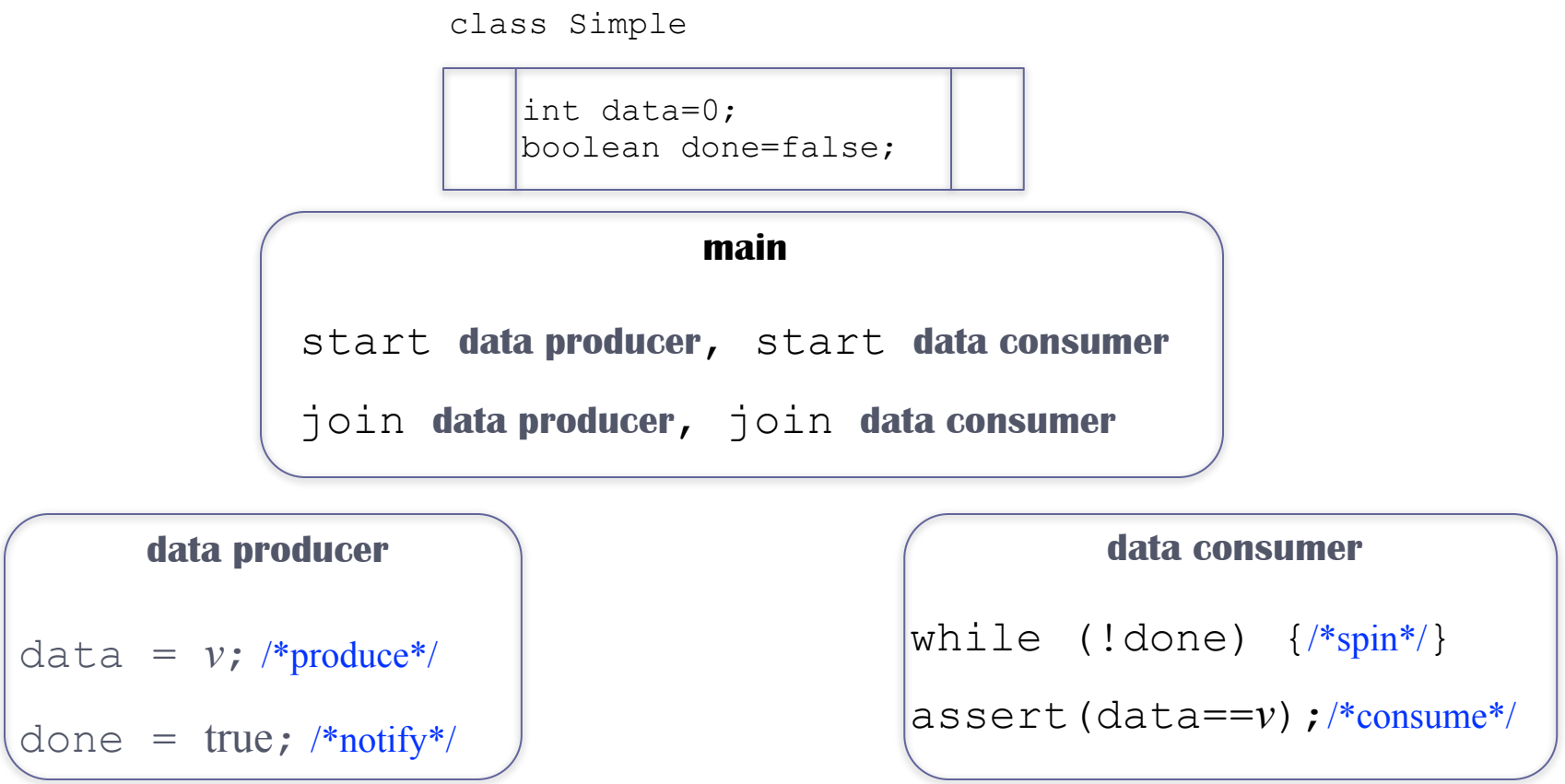
- In JMM, a situation that can lead to non-sequentially consistent behavior is a data race.

Java Memory Model

- Happens-before order in JMM
  - A transitive, irreflexive partial order on the actions in an execution
  - Transitive closure of the union of
    - ➡program order (intra-thread)
    - ➡synchronize-with order (inter-thread)

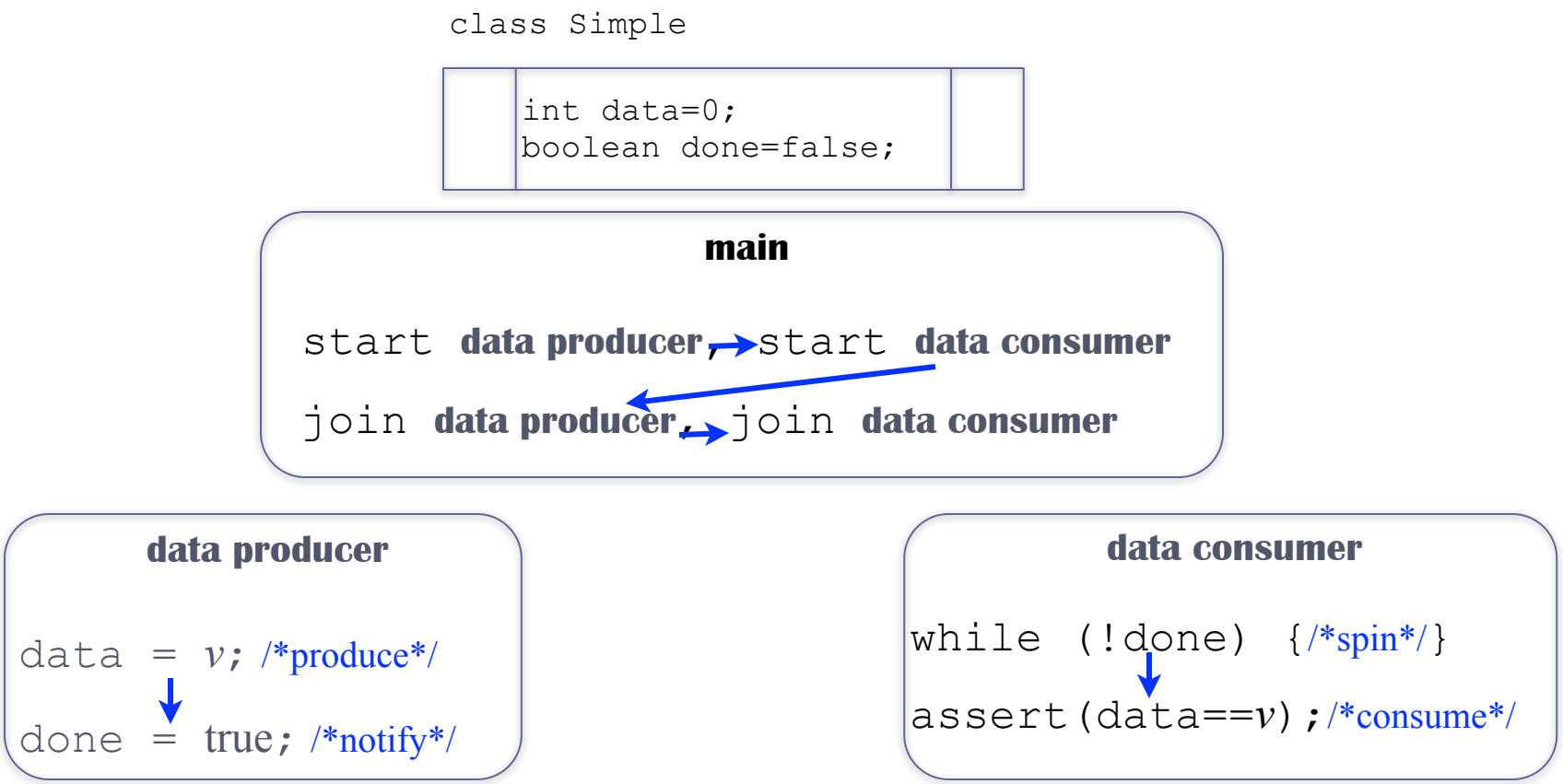
Java Memory Model

➡program order (intra-thread)



Java Memory Model

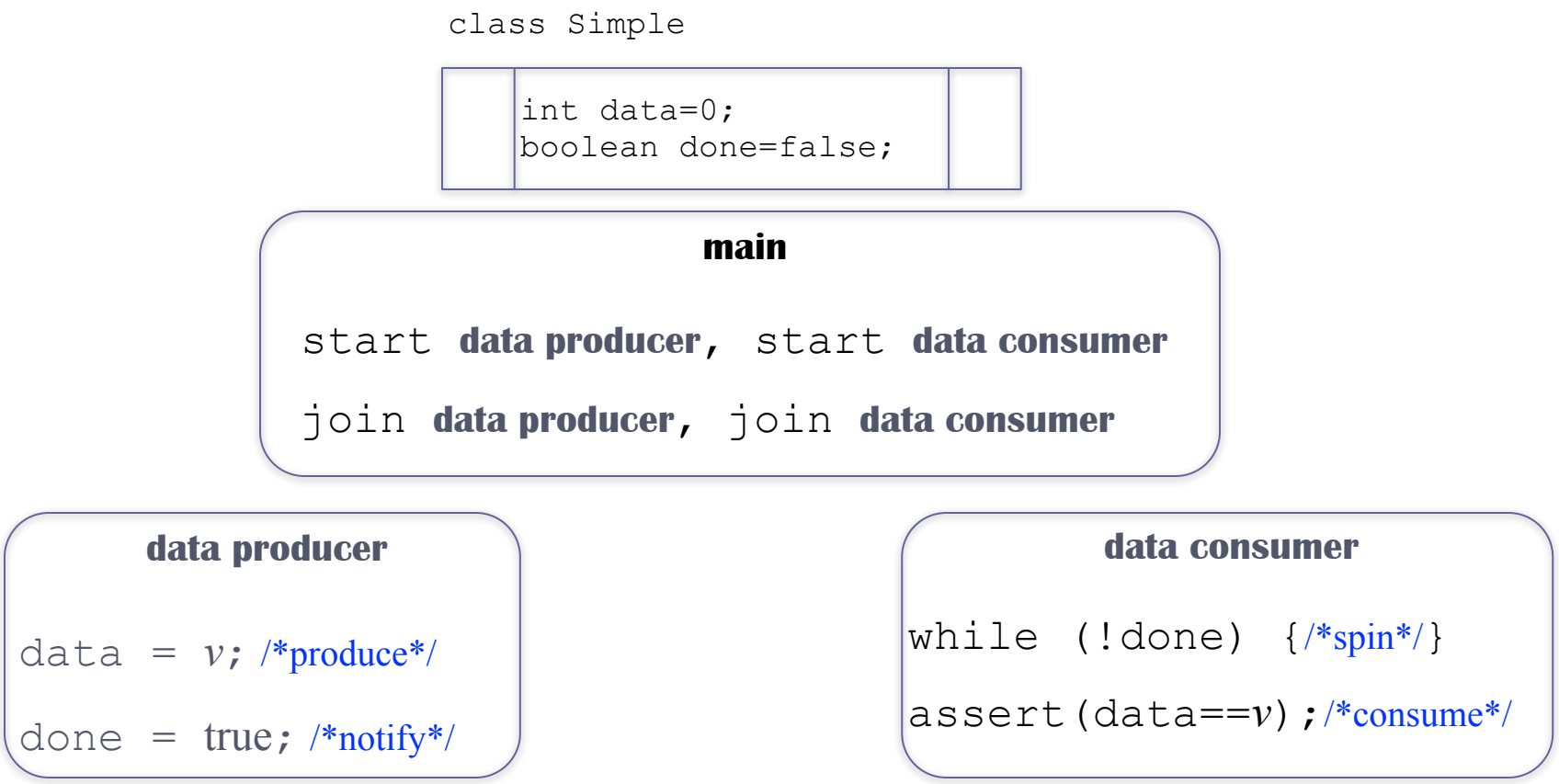
➡program order (intra-thread)



Java Memory Model

➡program order (intra-thread)

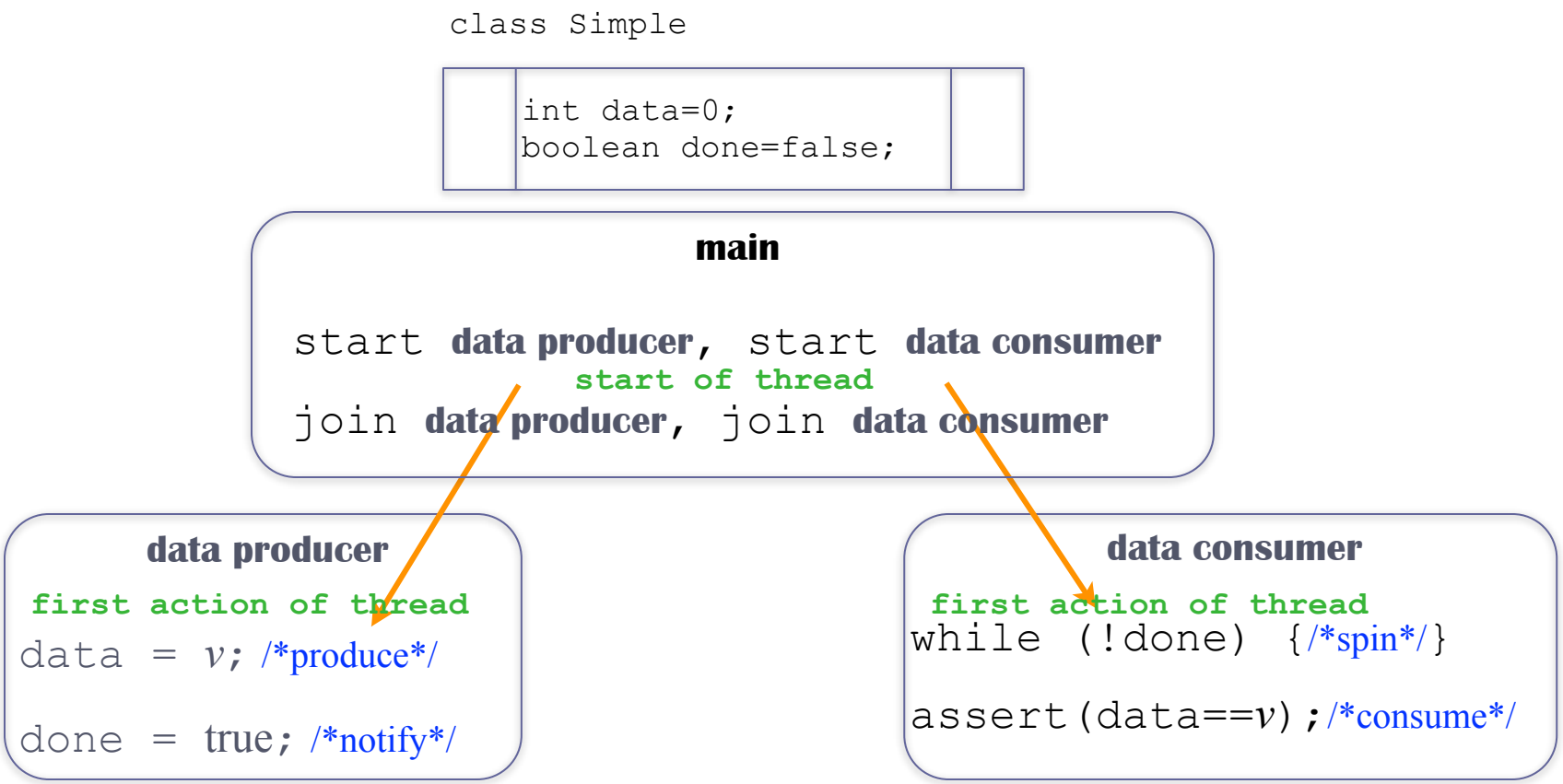
➡synchronize-with order (inter-thread)



Java Memory Model

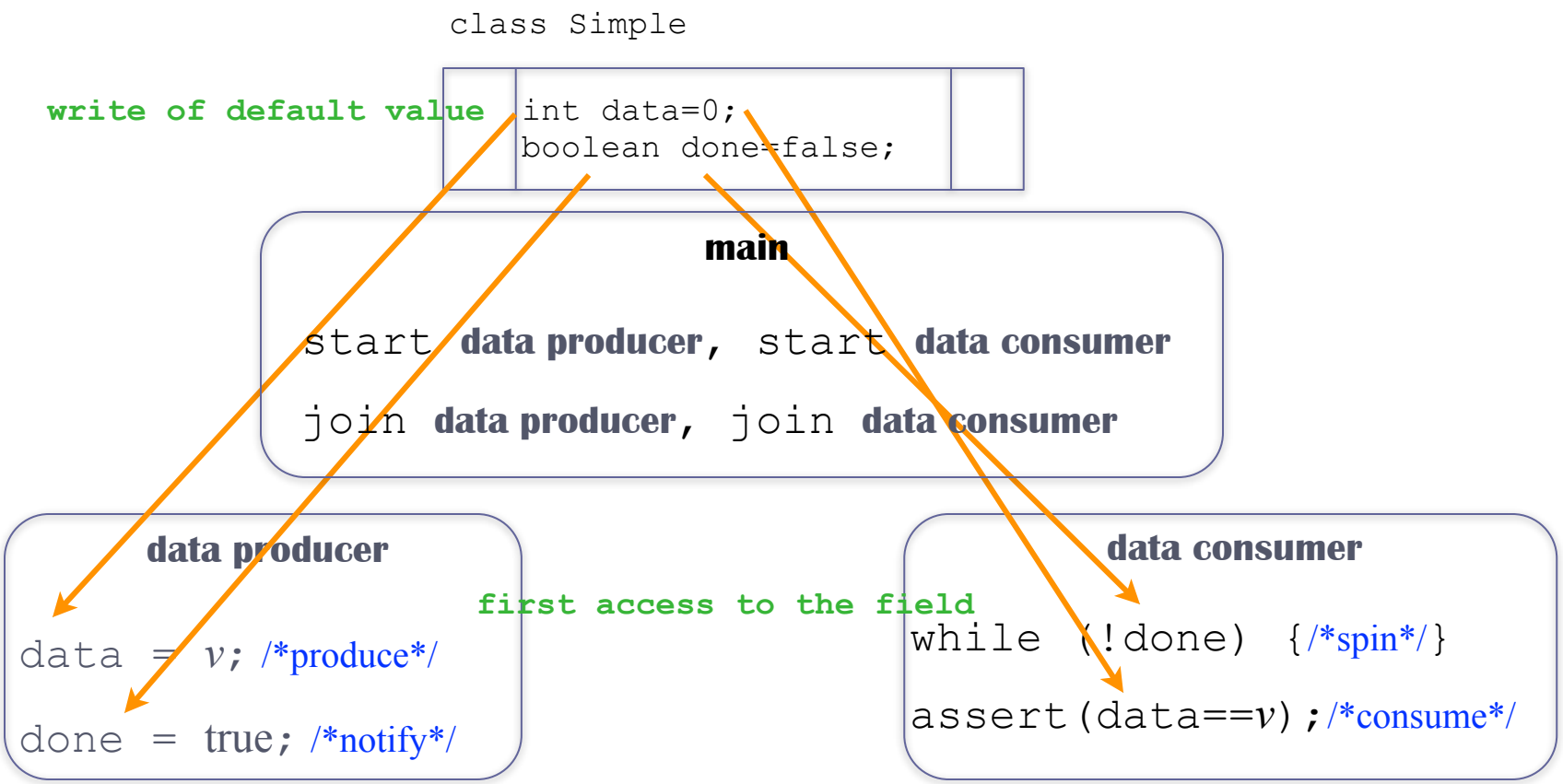
➡program order (intra-thread)

➡synchronize-with order (inter-thread)



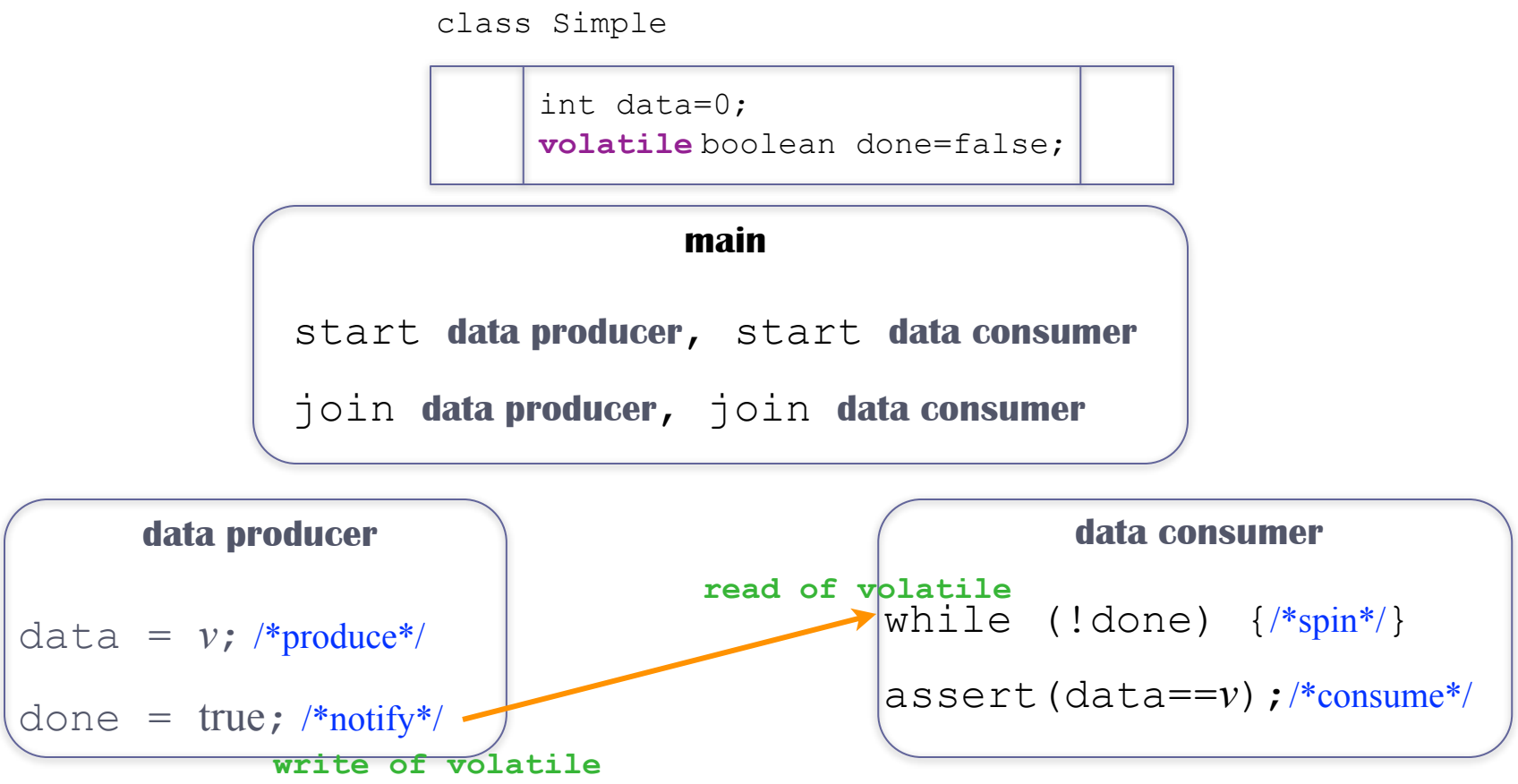
Java Memory Model

- ➡program order (intra-thread)
- ➡synchronize-with order (inter-thread)



Java Memory Model

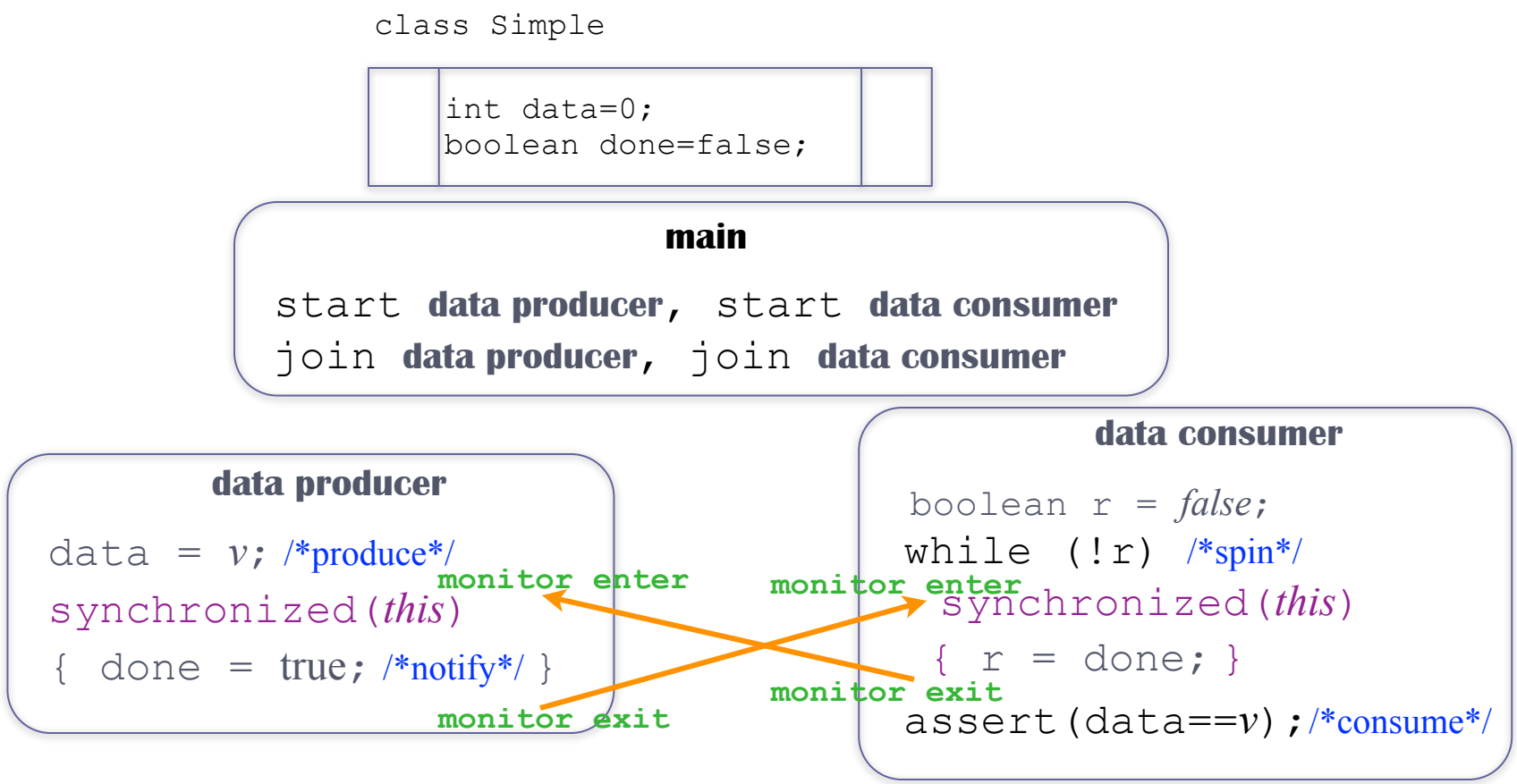
- ➡program order (intra-thread)
- ➡synchronize-with order (inter-thread)



Java Memory Model

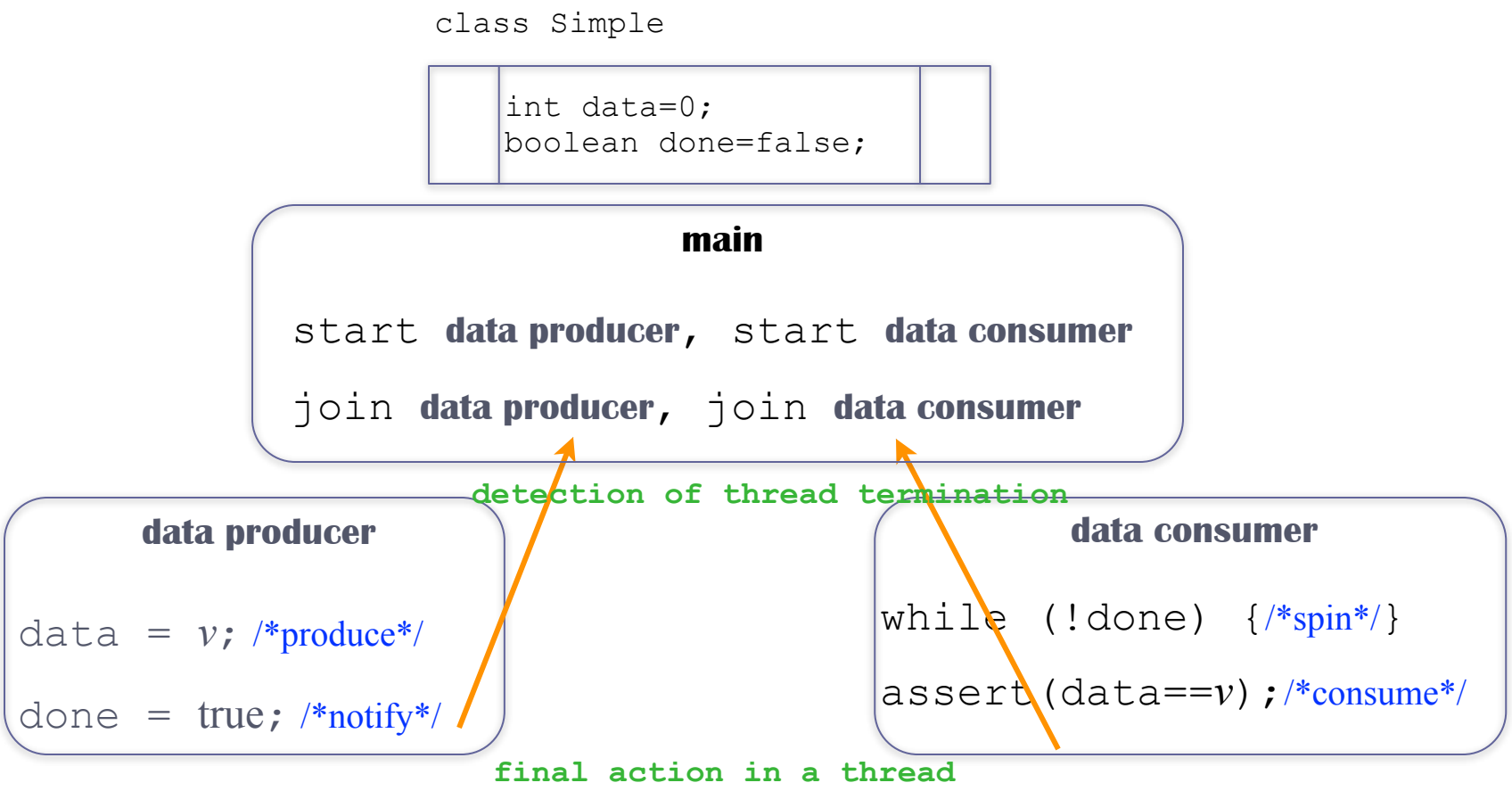
➡program order (intra-thread)

➡synchronize-with order (inter-thread)



Java Memory Model

- program order (intra-thread)
- synchronize-with order (inter-thread)

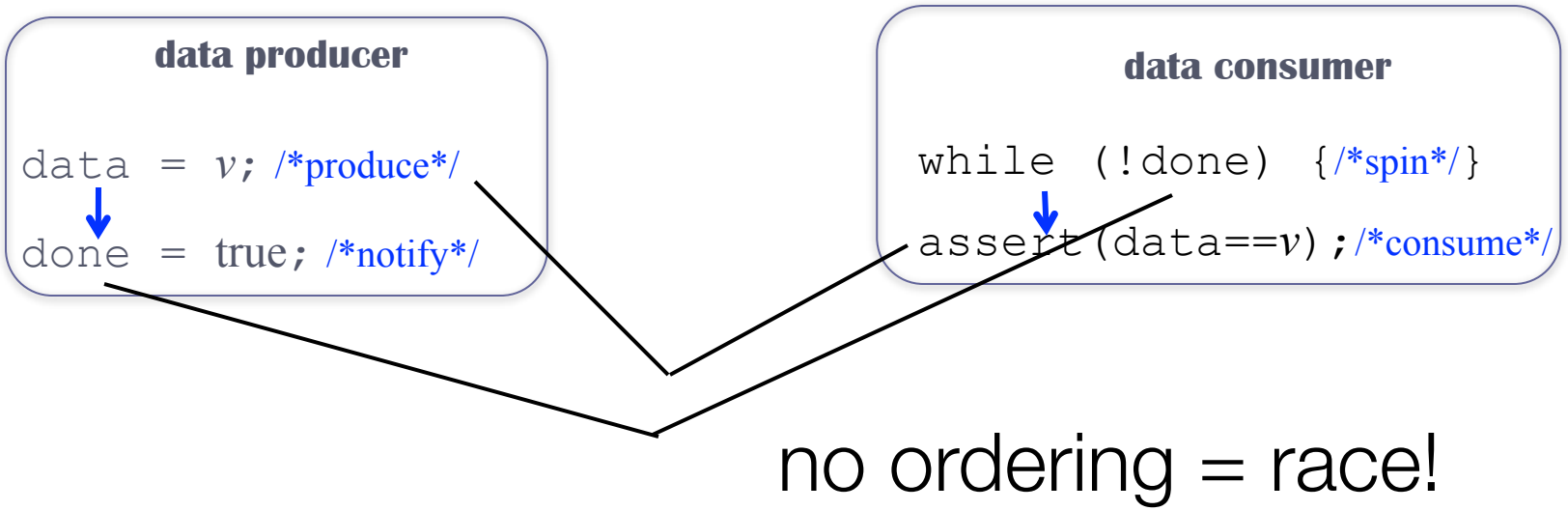


Java Memory Model

- Happens-before order in JMM
  - A transitive, irreflexive partial order on the actions in an execution
  - Transitive closure of the union of
    - ➡program order (intra-thread)
    - ➡synchronize-with order (inter-thread)
      1. An unlock action on a monitor lock *m* synchronizes-with all subsequent lock actions on *m* by any thread.
      2. A write to a volatile variable *v* synchronizes-with all subsequent reads of *v*.
      3. The action of starting a thread synchronizes-with the first action of the newly started thread.
      4. The final action in a thread synchronizes-with an action in any other thread that detects the thread's termination.
      5. The writing of default values of every object field synchronizes-with the first access of the field

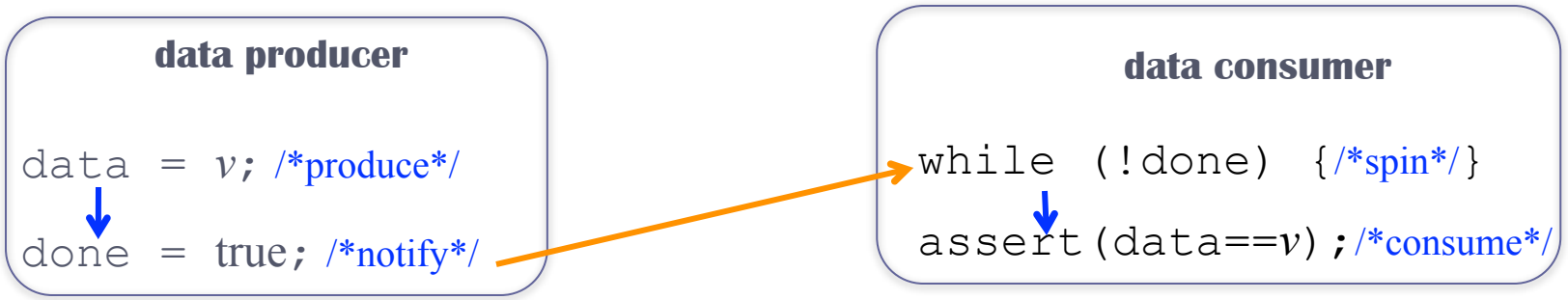
Java Memory Model

- Data Races in `Simple` class



Java Memory Model

- Data Races in `SimpleVolatile` class



no race!

Java Memory Model

- Java Memory Model formal definition

W : the write-seen function (read action -> write action)

- determines which write can be seen to each read

Java Memory Model

- Happens-before consistency

  : determines the values a non-volatile read can see

$$w = W(r)$$

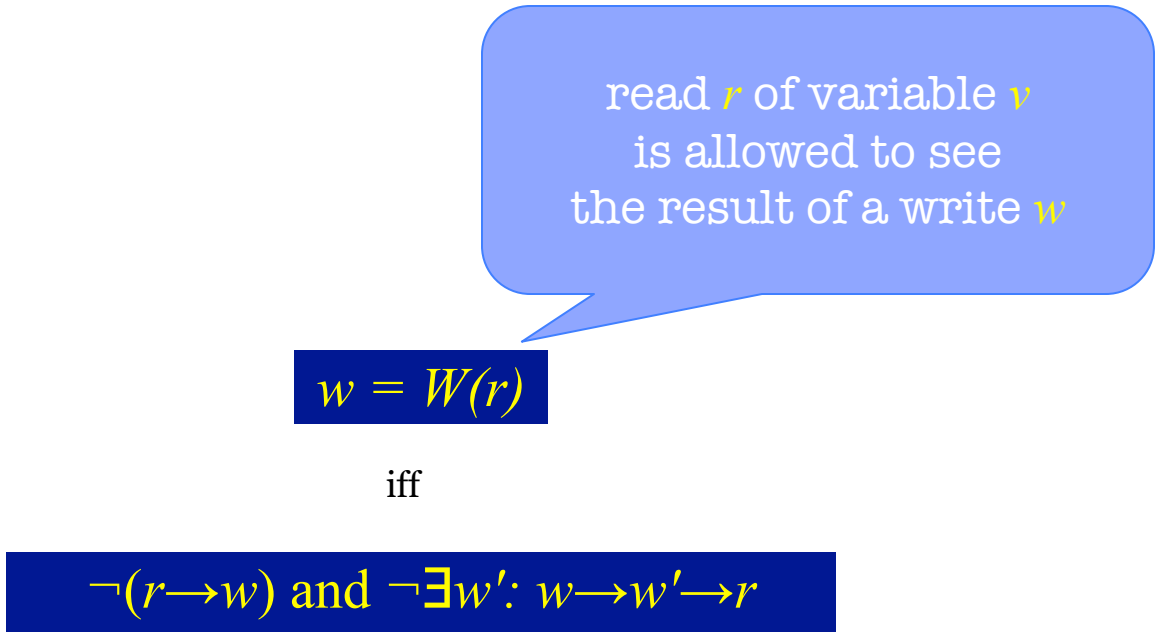
iff

$$\neg(r \rightarrow w) \text{ and } \neg\exists w' : w \rightarrow w' \rightarrow r$$

Java Memory Model

- Happens-before consistency

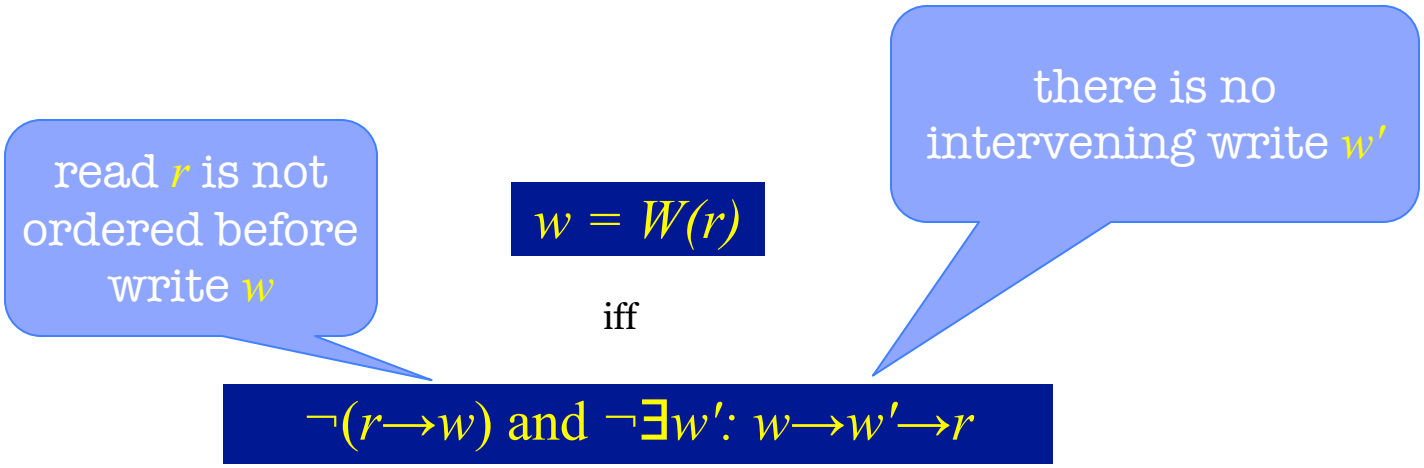
  : determines the values a non-volatile read can see



Java Memory Model

- Happens-before consistency

  : determines the values a non-volatile read can see



Java Memory Model

- Synchronization order consistency

  : determines the values a volatile read can see.

synchronization order is *consistent with* **program order**

each read of a volatile sees the **last write** to the variable to come before it in the synchronization order

Java Memory Model

- Fundamental property of Java Memory Model

Programs whose SC executions have no races

must have only SC executions.

Java Memory Model

- Fundamental property of Java Memory Model

Programs whose SC executions have no races

must have only SC executions.

The data race freedom of a program
can be proved by verifying
all SC executions are free of data races.

Java Memory Model

- Fundamental property of Java Memory Model

Programs whose SC executions have no races

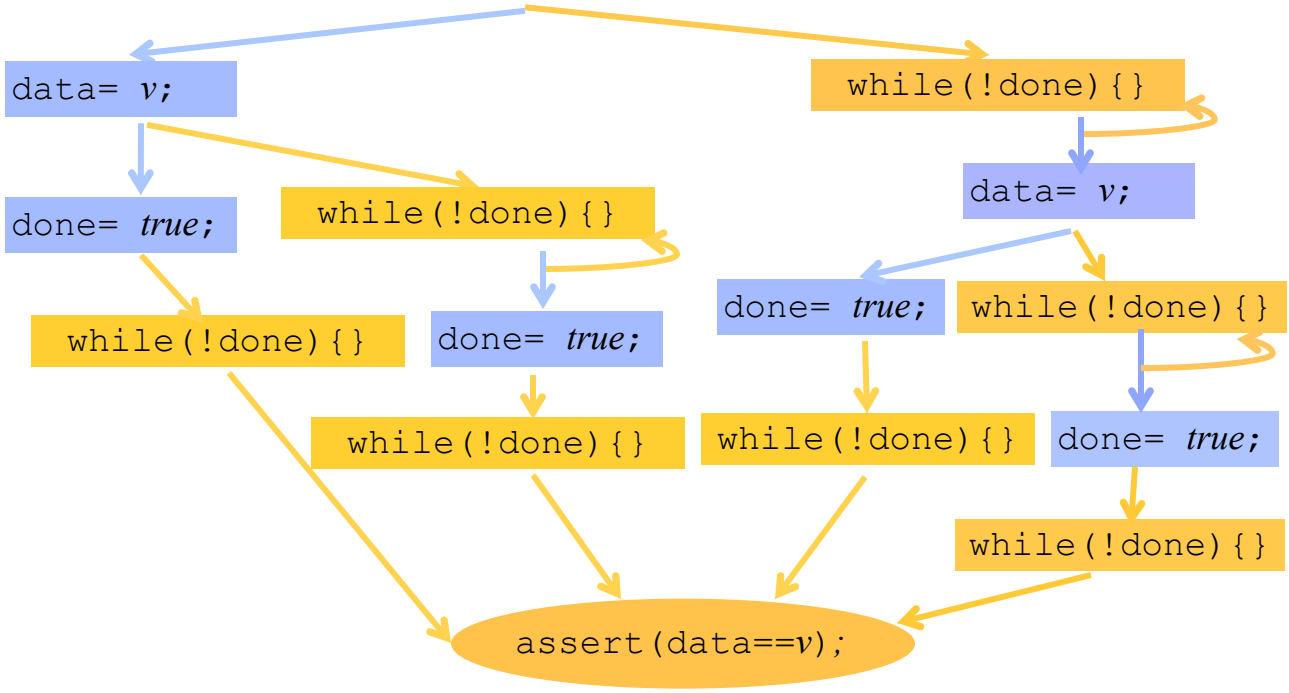
must have only SC executions.

The data race freedom of a program
can be proved by verifying
all SC executions are free of data races.

This justifies the use of JPF to find
a data race.

Java Memory Model

- Sequentially Consistent Executions of `SimpleVolatile` class



JRF approach

Data Race Detection

Algorithm

- Summary Function h

Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

$$h(v) = X$$

Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

$$h(v) = X$$

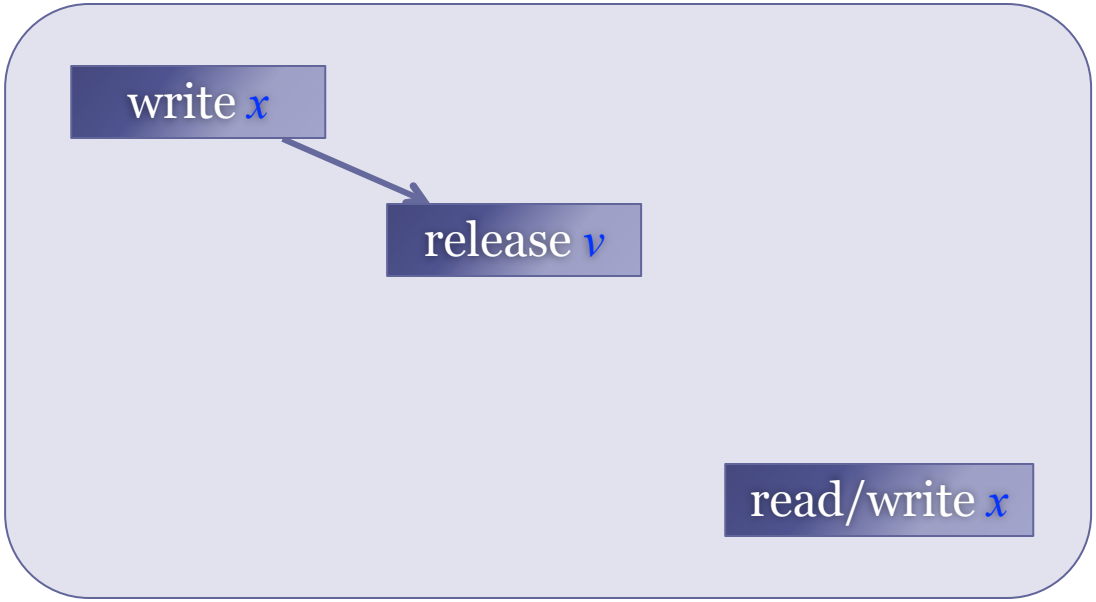
a memory location either a volatile, a thread, or a synchronization object

a set of memory locations whose recent update happens-before the release of v

Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

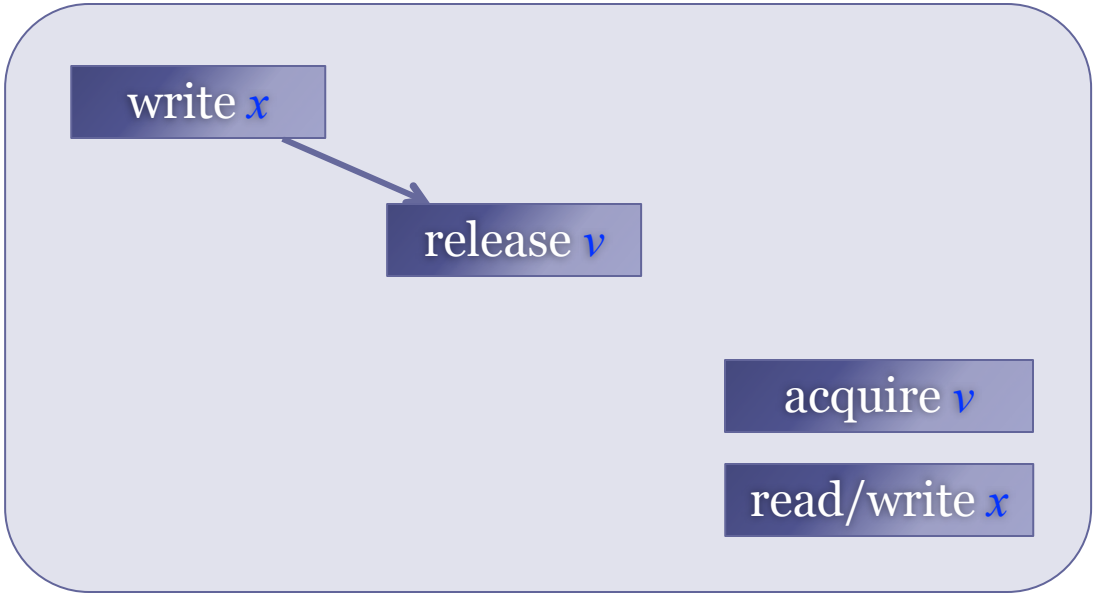
$$h(v) = X$$



Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

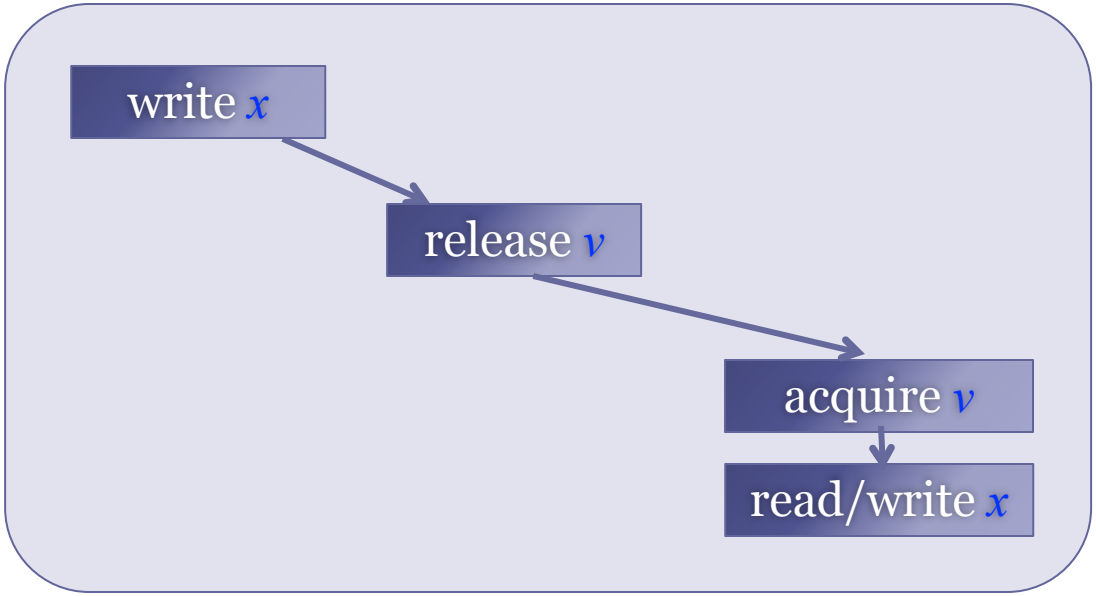
$$h(v) = X$$



Algorithm

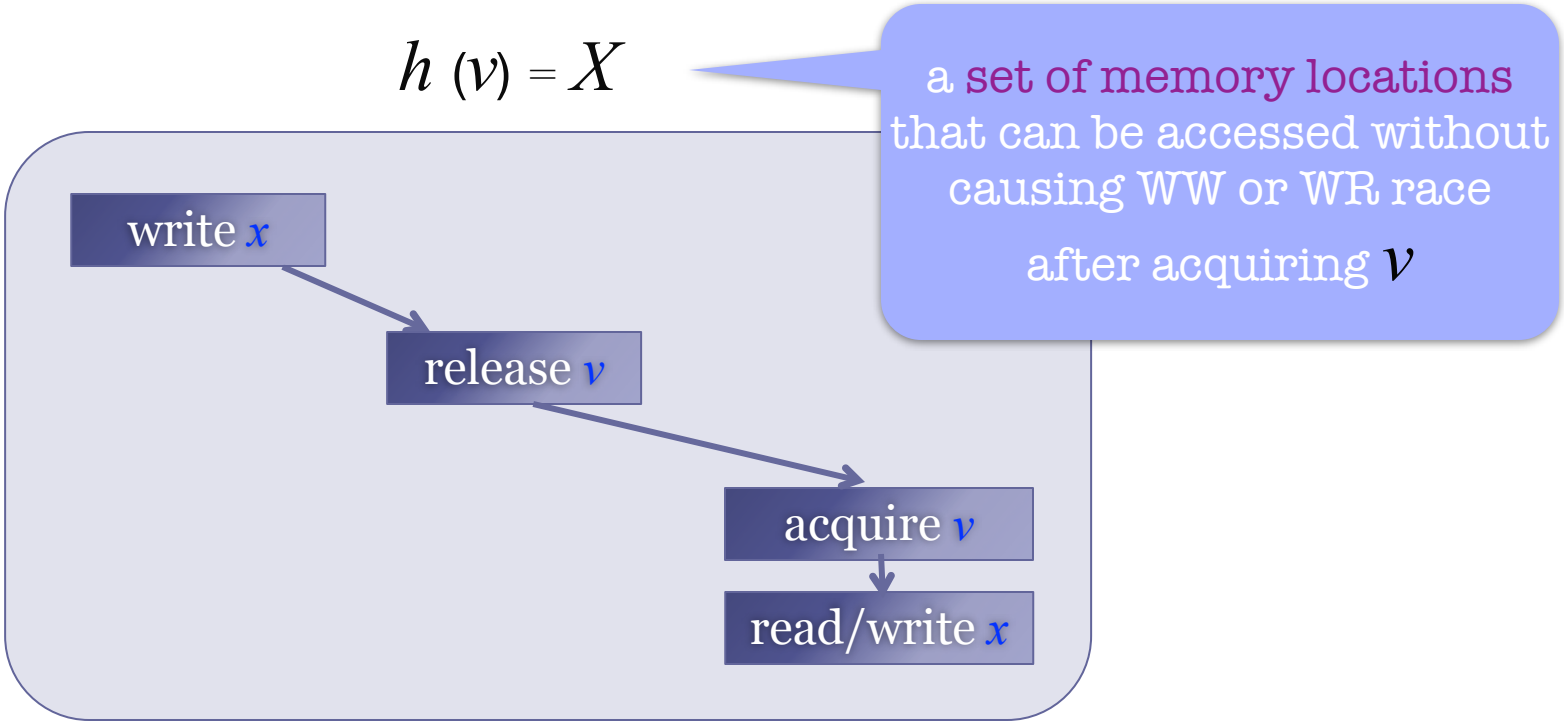
- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

$$h(v) = X$$



Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function



Algorithm

- Summary Function h
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

$$h(v) = X$$

$x \in h(t)$: thread t can access (read or write) non-volatile variable x without causing a WW(write-write) or WR(write-read) data

Algorithm

- Summary Function h

$$h : SynchAddr \cup Threads \rightarrow 2^{Addr}$$

acquire

release

invalidate

norace

Algorithm

- Summary Function h

$$h : SynchAddr \cup Threads \rightarrow 2^{Addr}$$

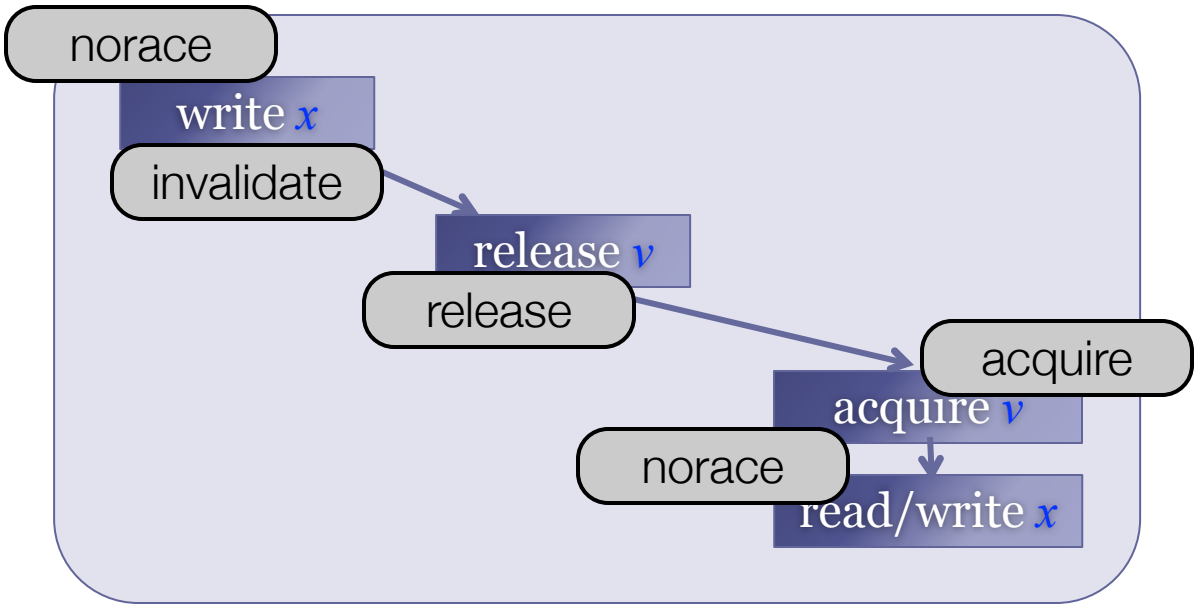
acquire	: volatile read, lock, ...
release	:volatile write, unlock, ...
invalidate	:non-volatile write
norace	: non-volatile read/write

Data Race Detection

Algorithm

- Summary Function h

$$h : SynchAddr \cup Threads \rightarrow 2^{Addr}$$



Algorithm

- Summary Function h
  - volatile write, lock, start a thread, ... : release
  - volatile read, unlock, join thread, ... : acquire
  - non-volatile write : norace & invalidate
  - non-volatile read : norace

Algorithm

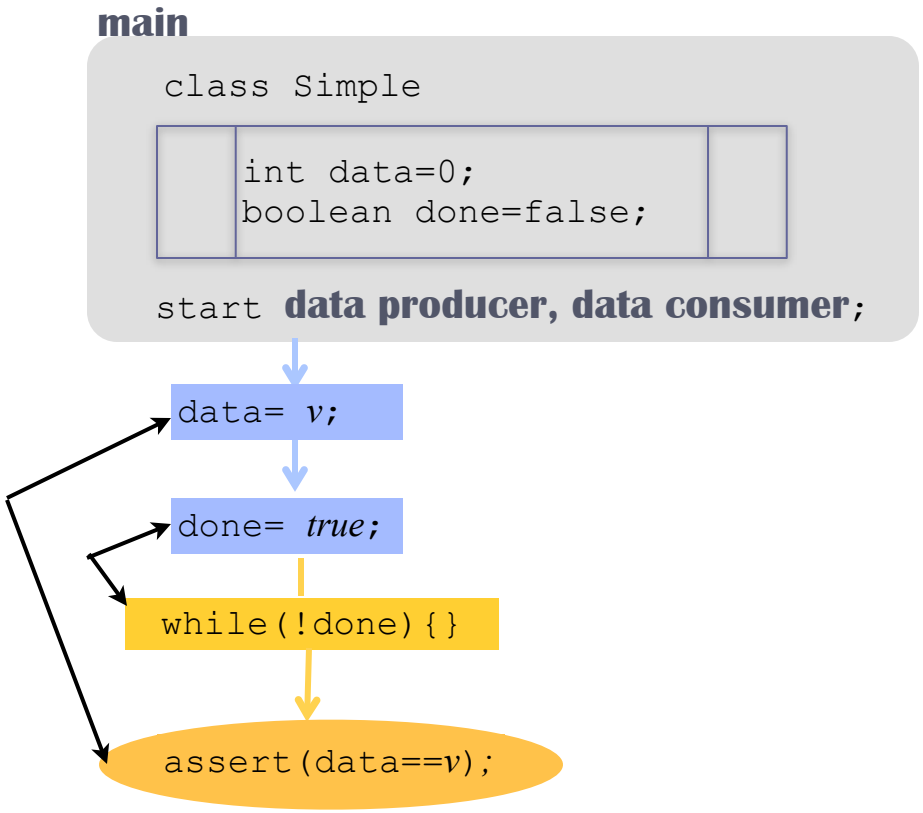
- Summary Function h
  - volatile read, lock, start a thread, ... : release
  - volatile write, unlock, join thread, ... : acquire
  - non-volatile write : norace & invalidate
  - non-volatile read : norace

- Maintain h and
  check *norace(x,t)* before reading or writing of non-volatile x by thread t.

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

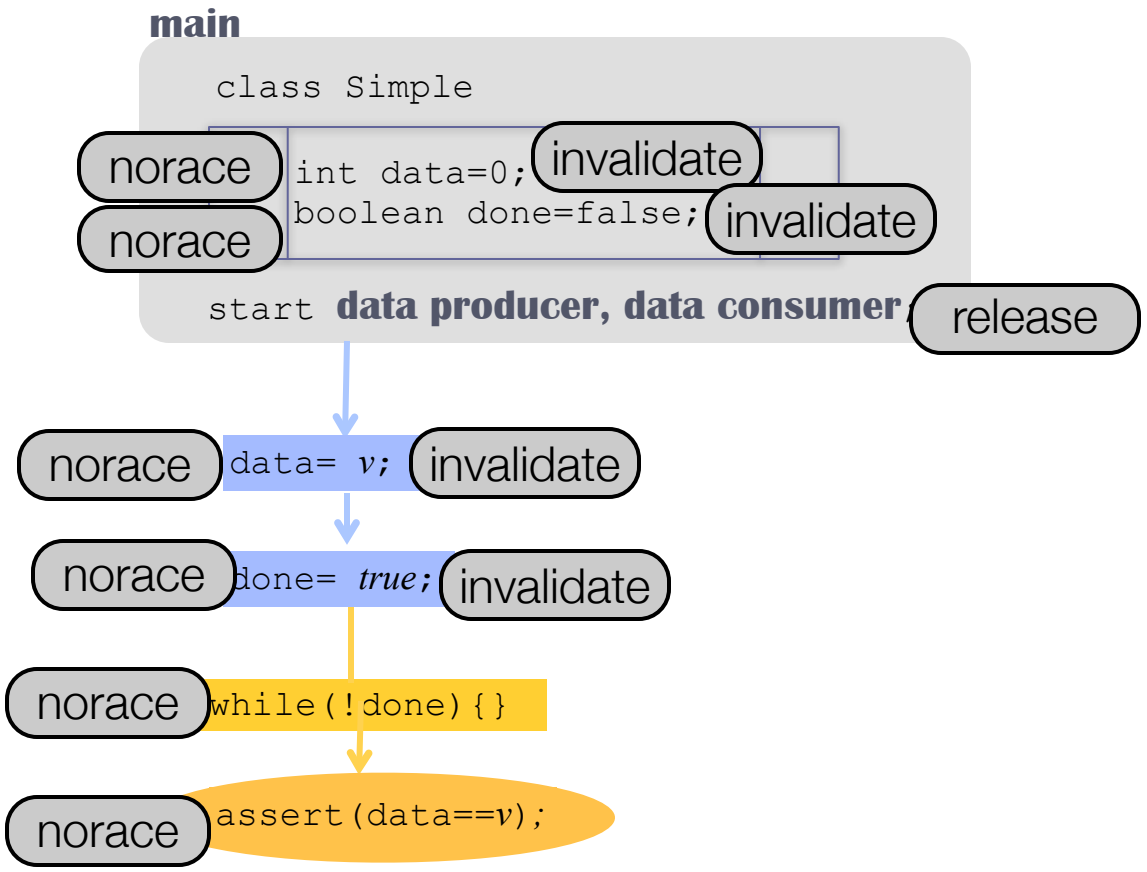


data race

Data Race Detection

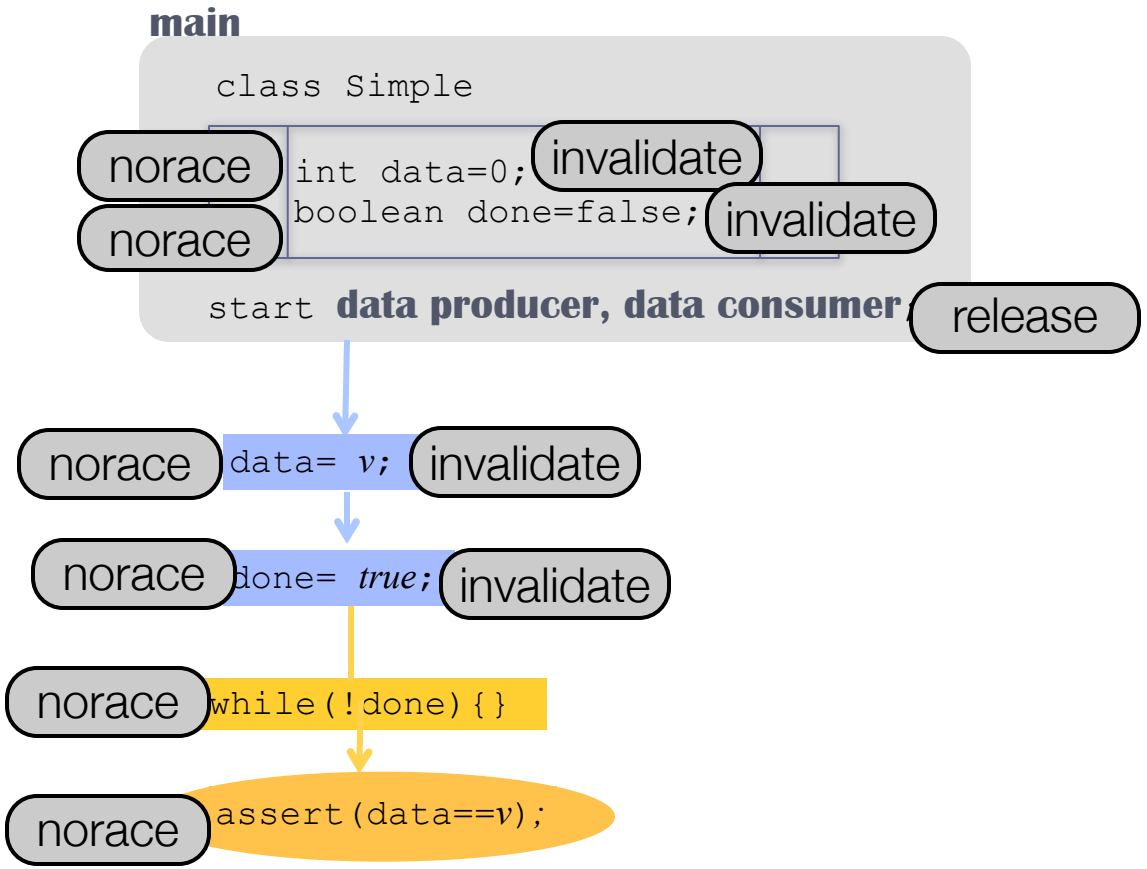
Algorithm

- Data race detection using summary function h for `Simple` java program



Algorithm

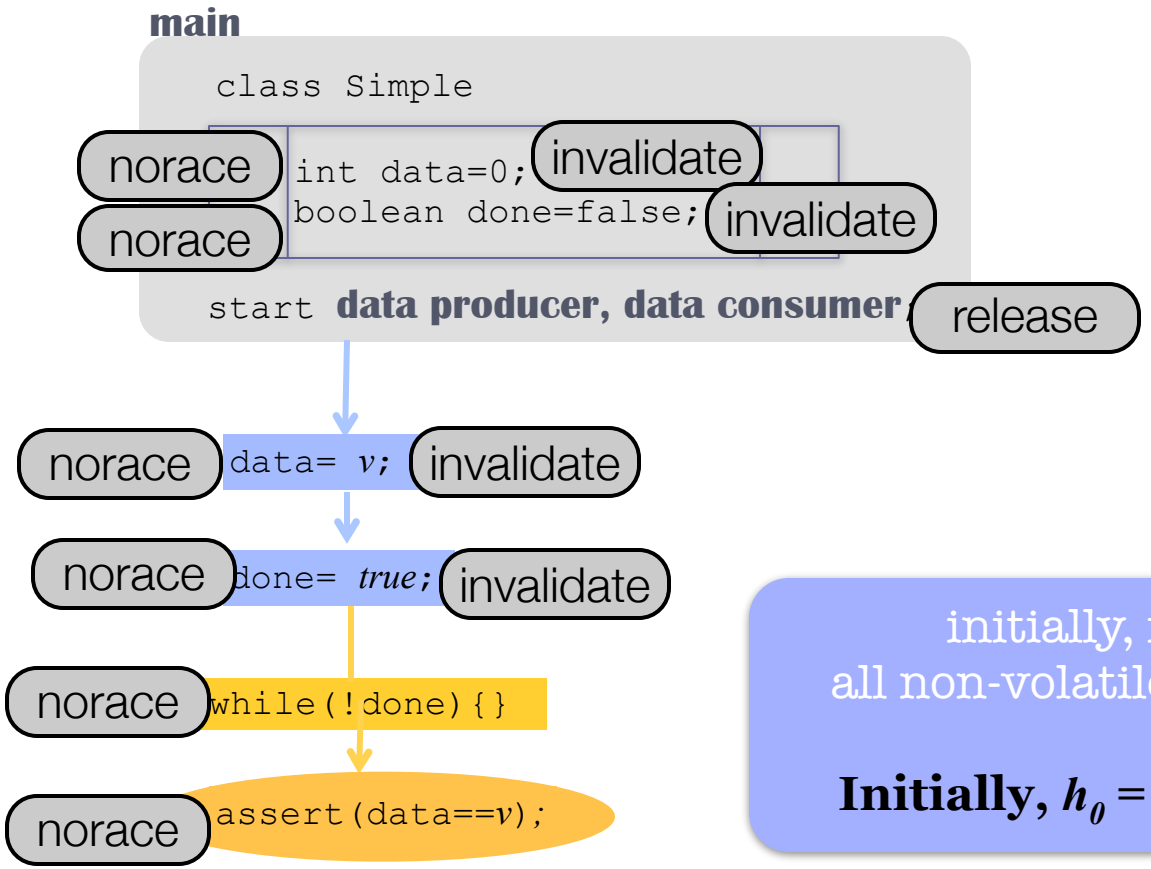
- Data race detection using summary function h for `Simple` java program



h(**main**)	
h(**data producer**)	
h(**data consumer**)	

Algorithm

- Data race detection using summary function h for `Simple` java program

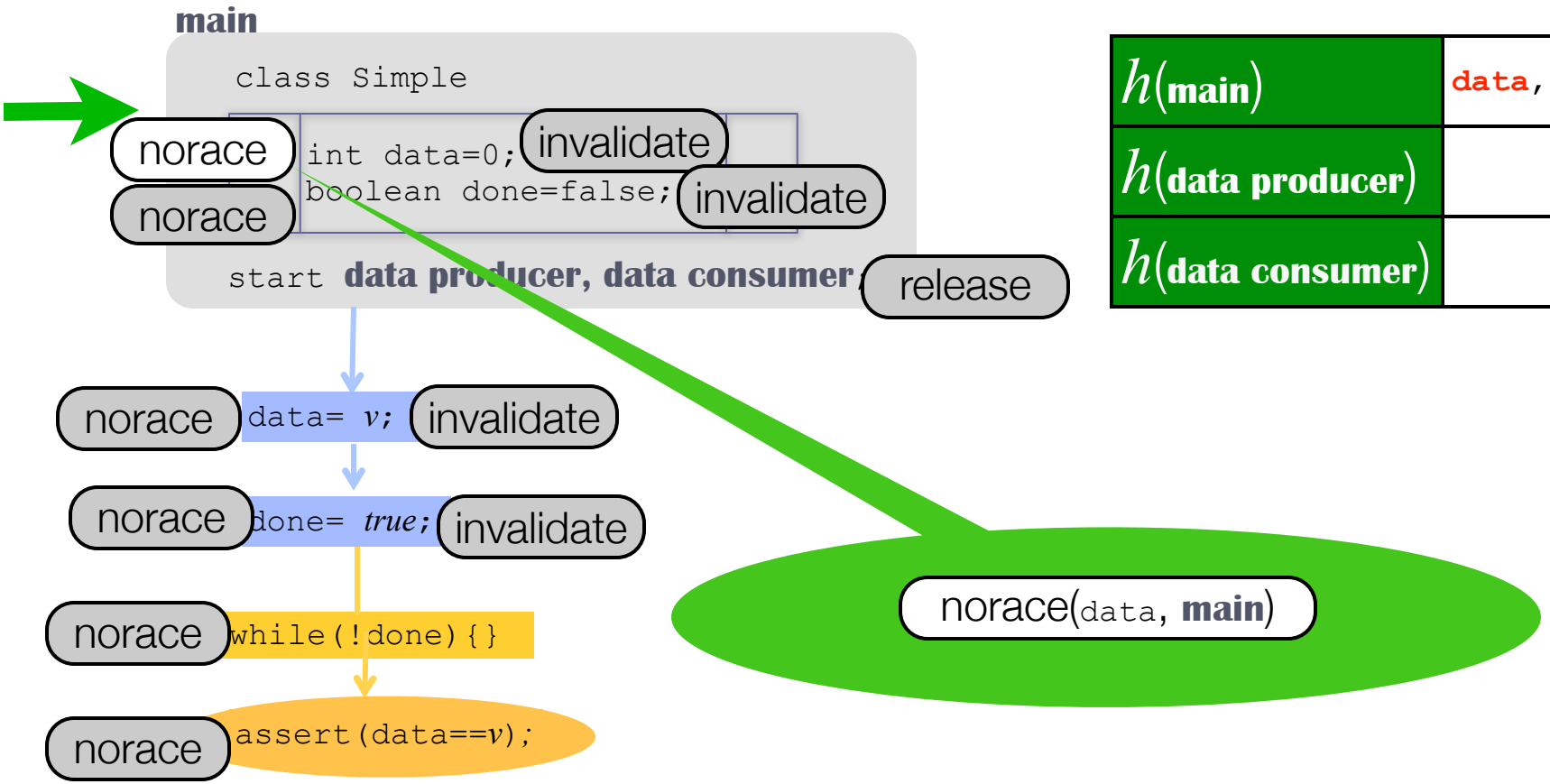


h(**main**)	`data, done`
h(**data producer**)	
h(**data consumer**)	

initially, main thread can access
all non-volatile static fields of the program
without a race
Initially, $h_0 = \lambda z.$**if** z=*main* **then** *static(P)* **else** $\perp$

Algorithm

- Data race detection using summary function h for `Simple` java program

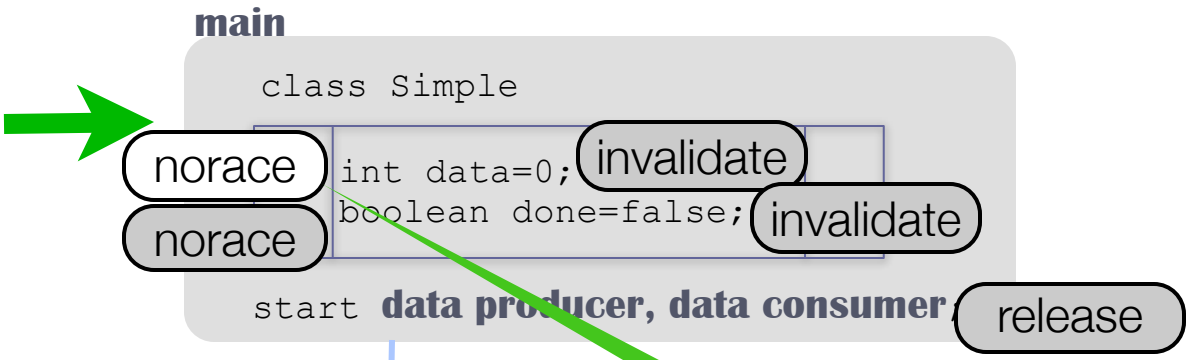


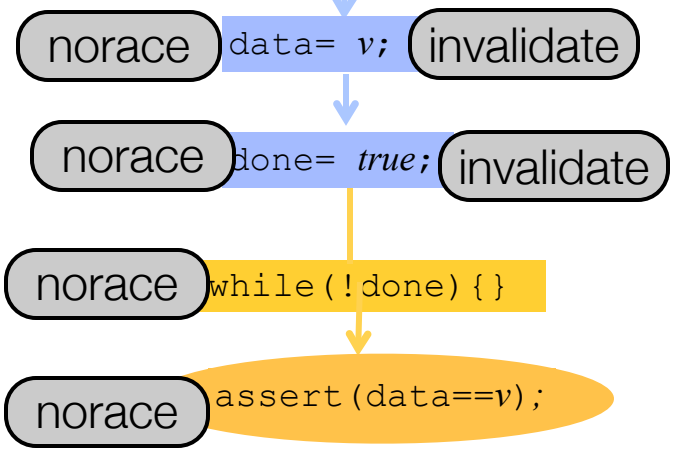
h(main)	data, done
h(data producer)	
h(data consumer)	

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program





h(**main**)	`data`, `done`
h(**data producer**)	
h(**data consumer**)	

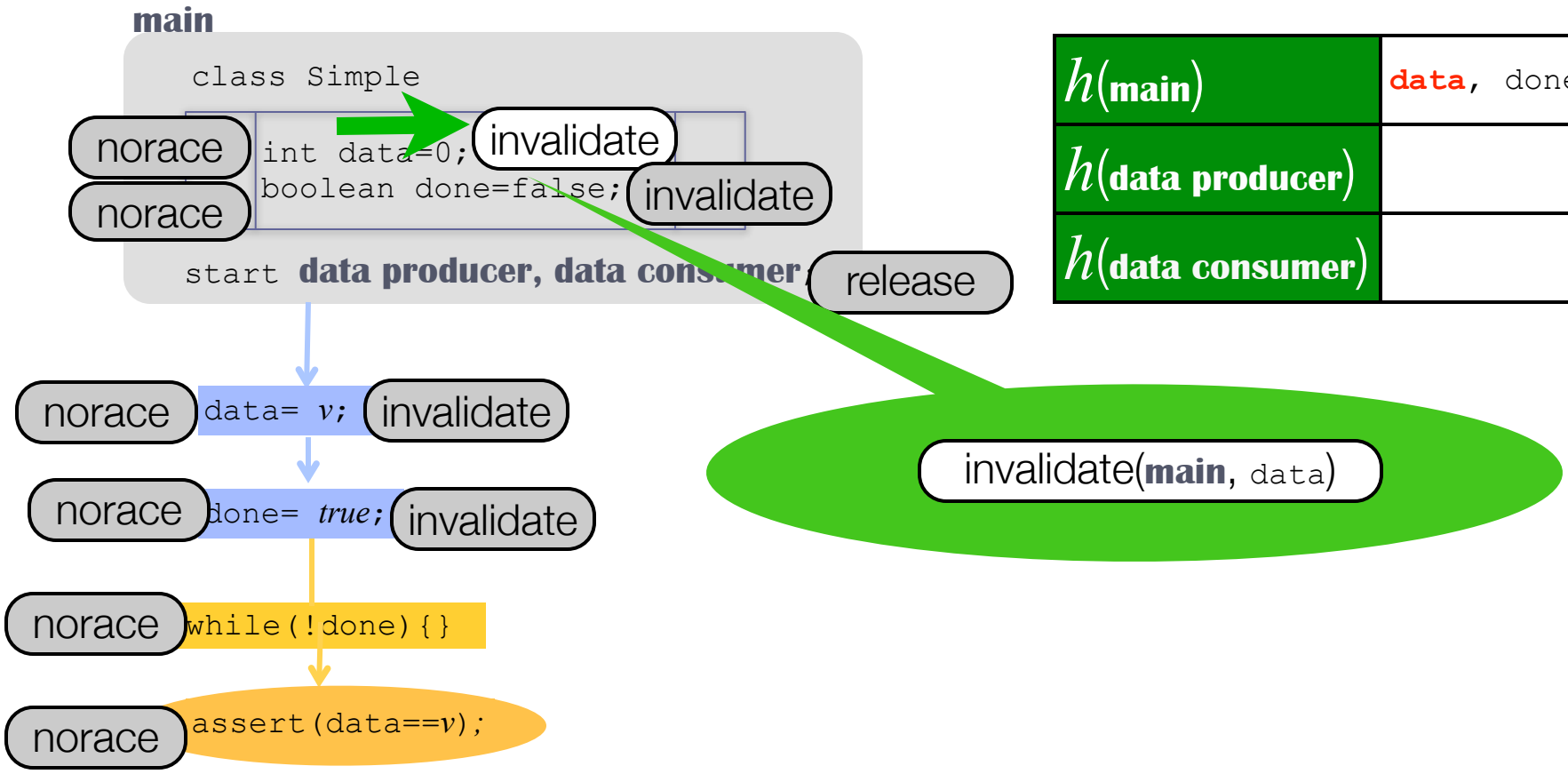
norace checks the accessibility

norace(x, t) = x ∈ h(t)

Data Race Detection

Algorithm

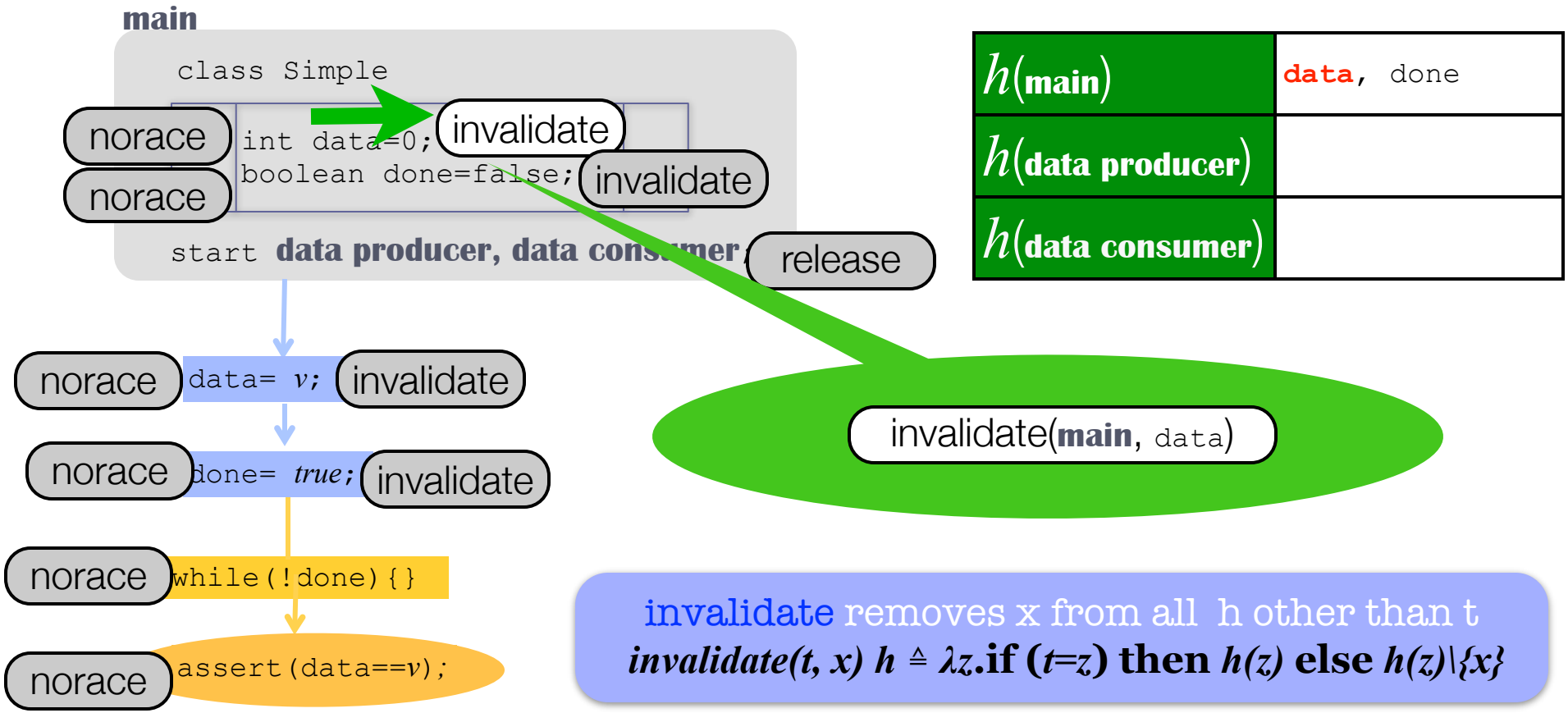
- Data race detection using summary function h for `Simple` java program



Data Race Detection

Algorithm

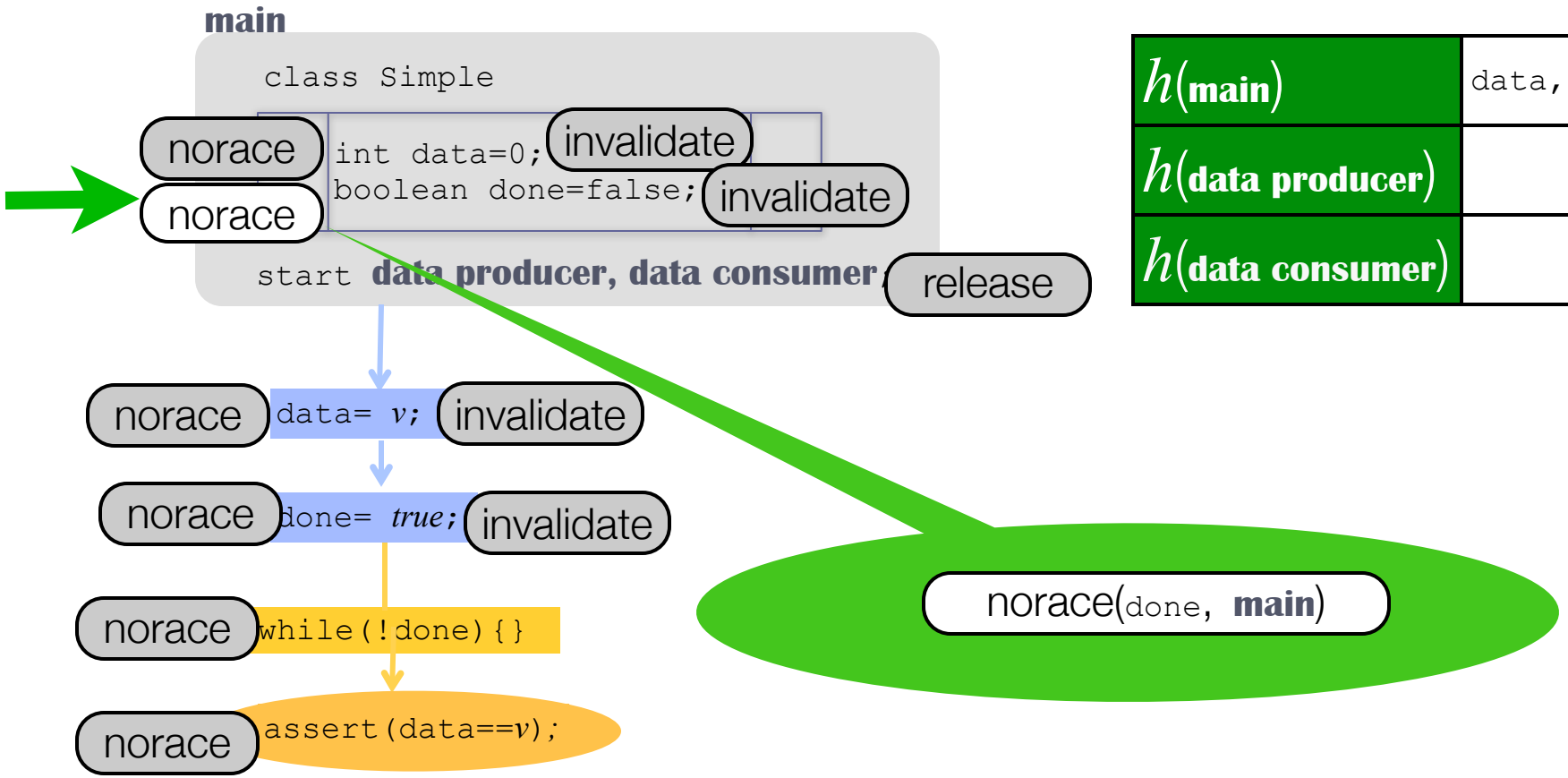
- Data race detection using summary function h for `Simple` java program



Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

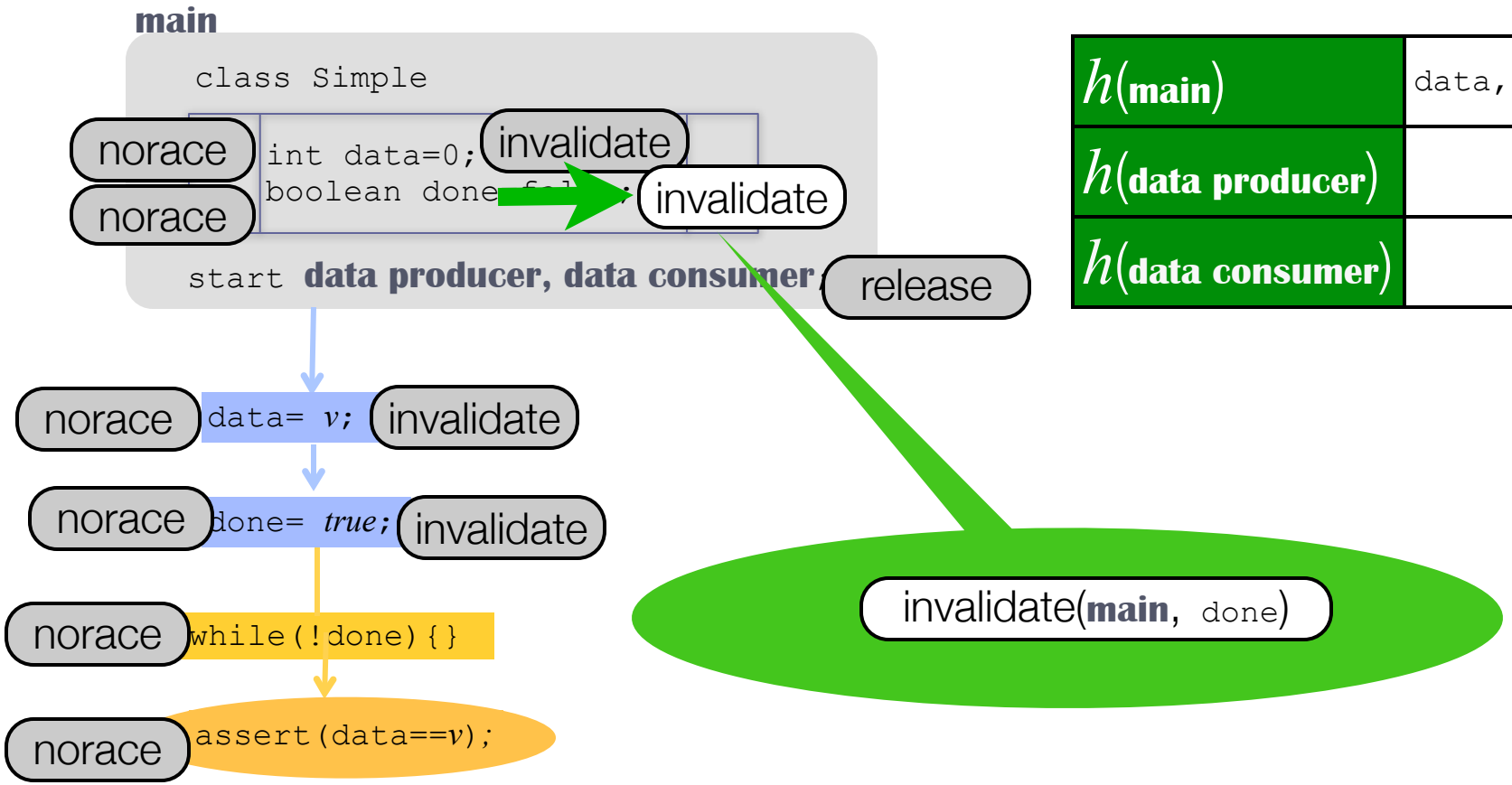


h(main)	data, done
h(data producer)	
h(data consumer)	

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

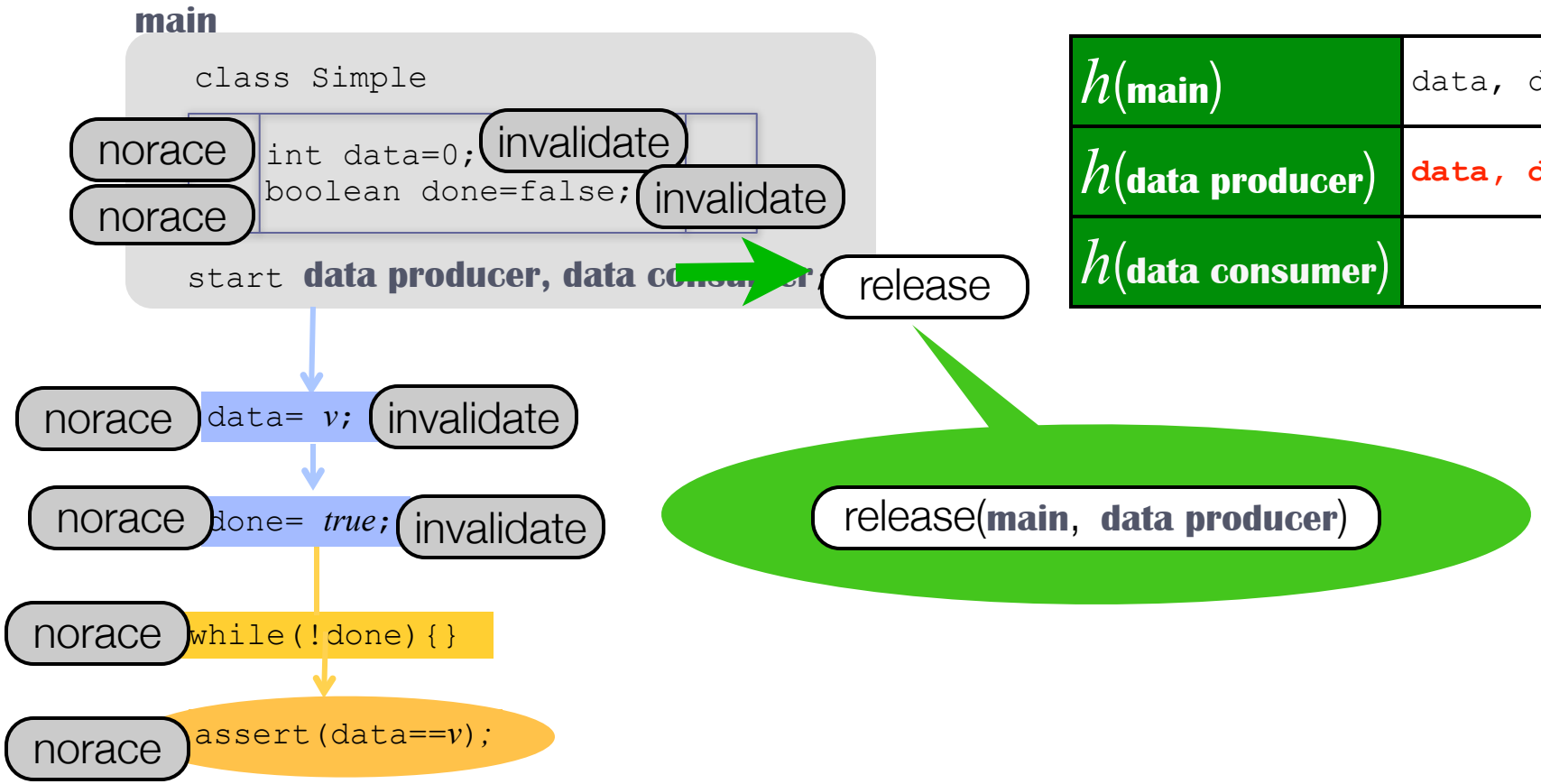


h(**main**)	`data`, **done**
h(**data producer**)	
h(**data consumer**)	

Data Race Detection

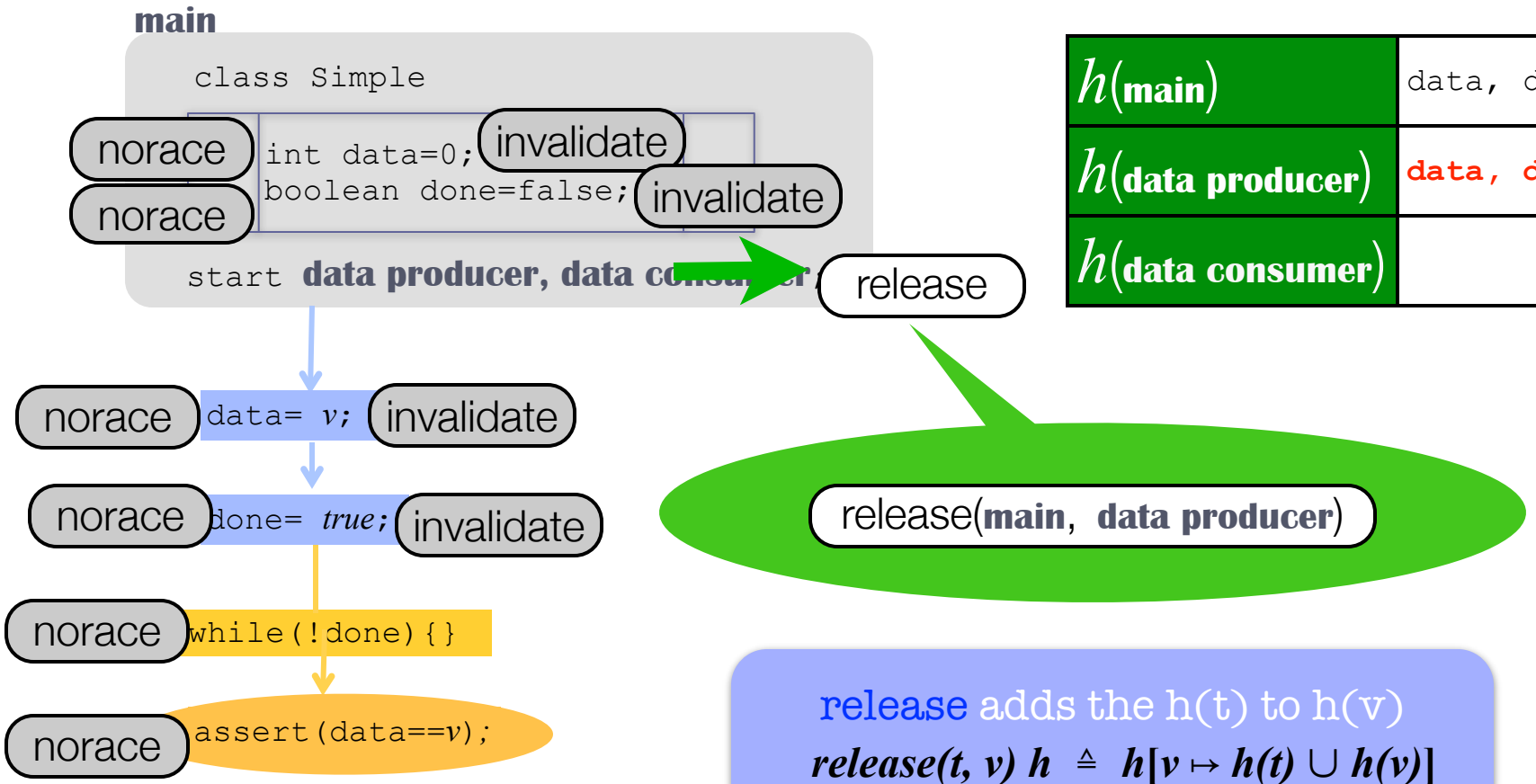
Algorithm

- Data race detection using summary function h for `Simple` java program



Algorithm

- Data race detection using summary function h for `Simple` java program



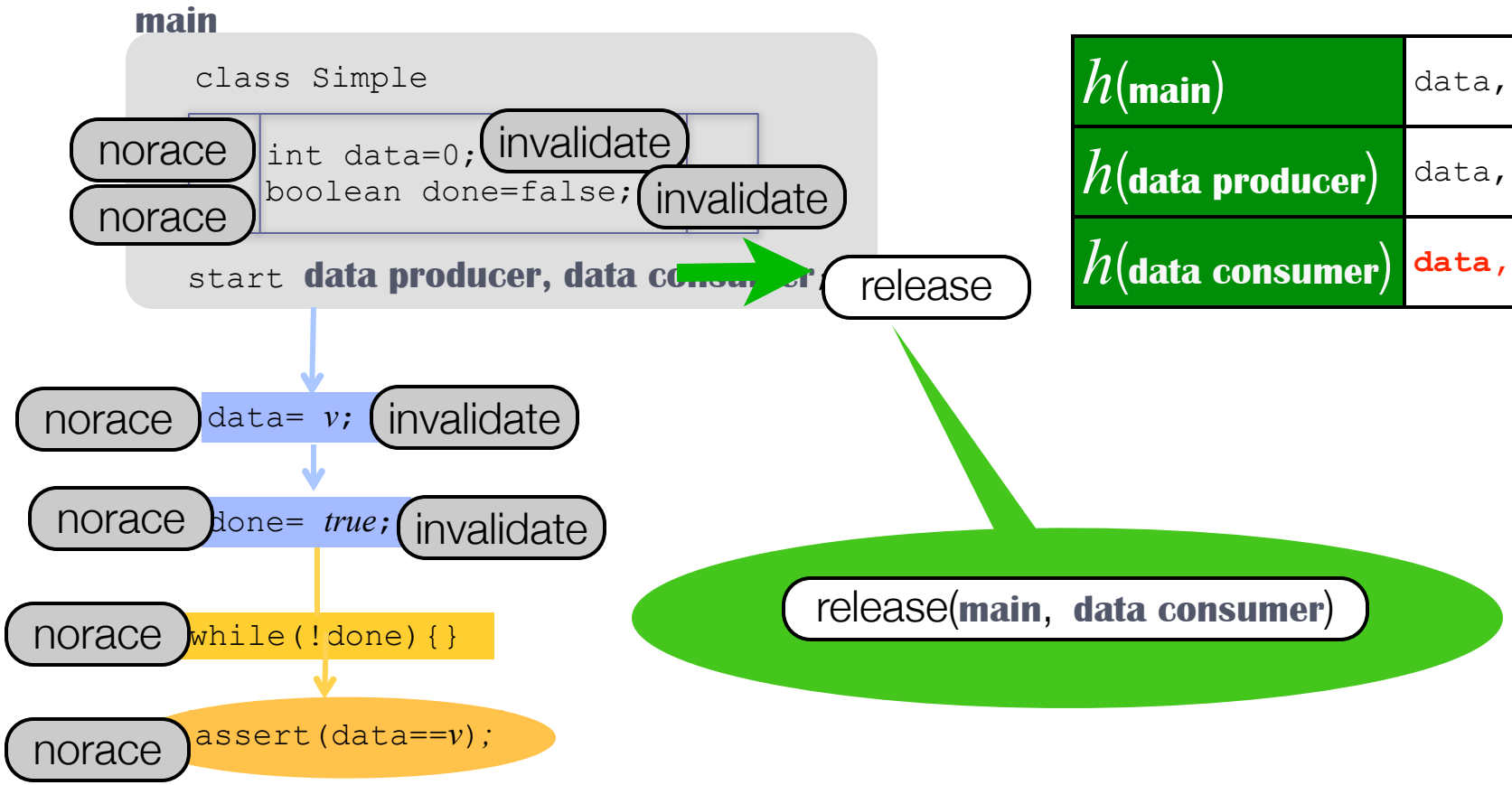
release adds the h(t) to h(v)

$$release(t, v)\ h \triangleq h[v \mapsto h(t) \cup h(v)]$$

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

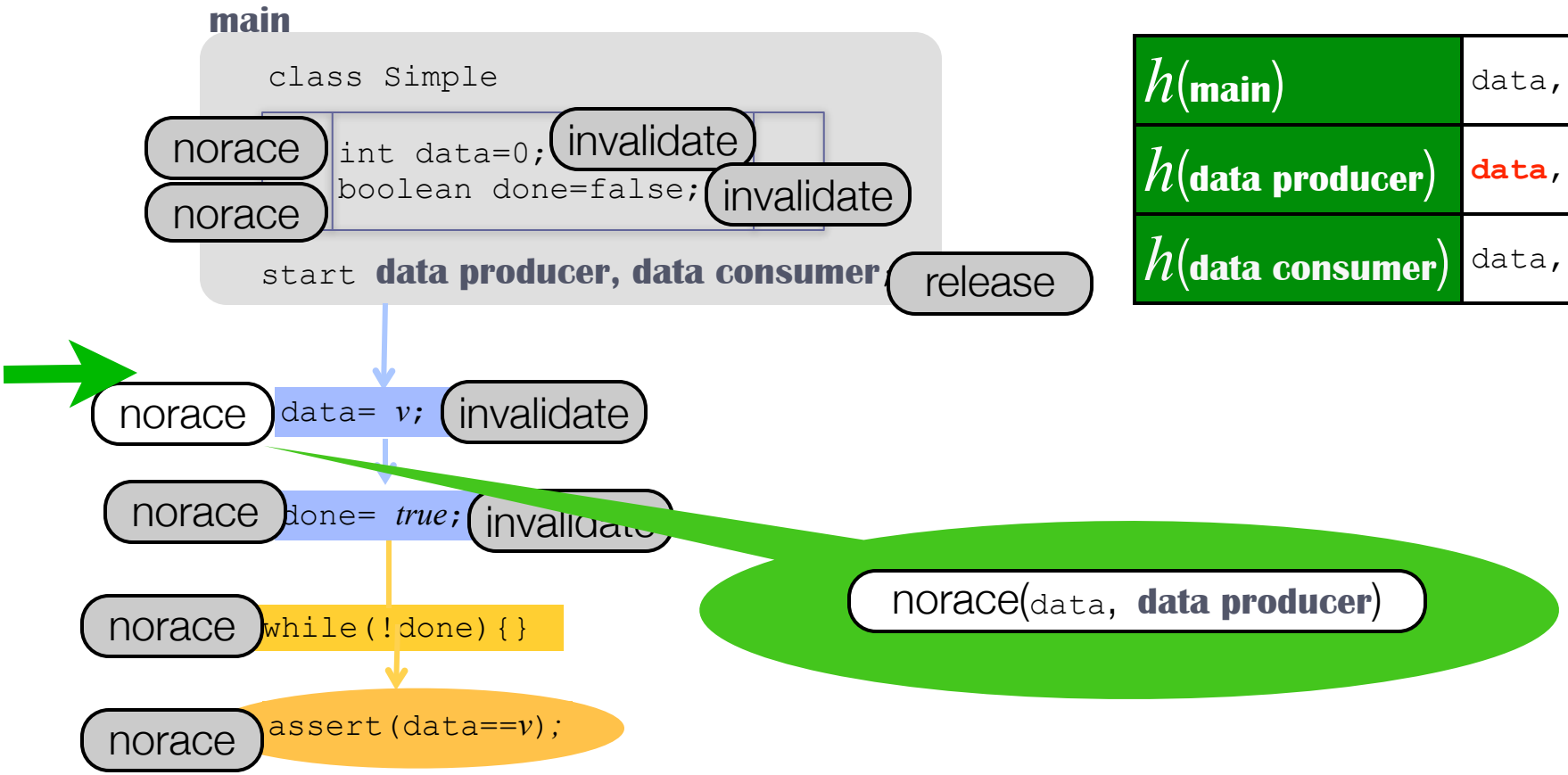


h(main)	data, done
h(data producer)	data, done
h(data consumer)	data, done

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

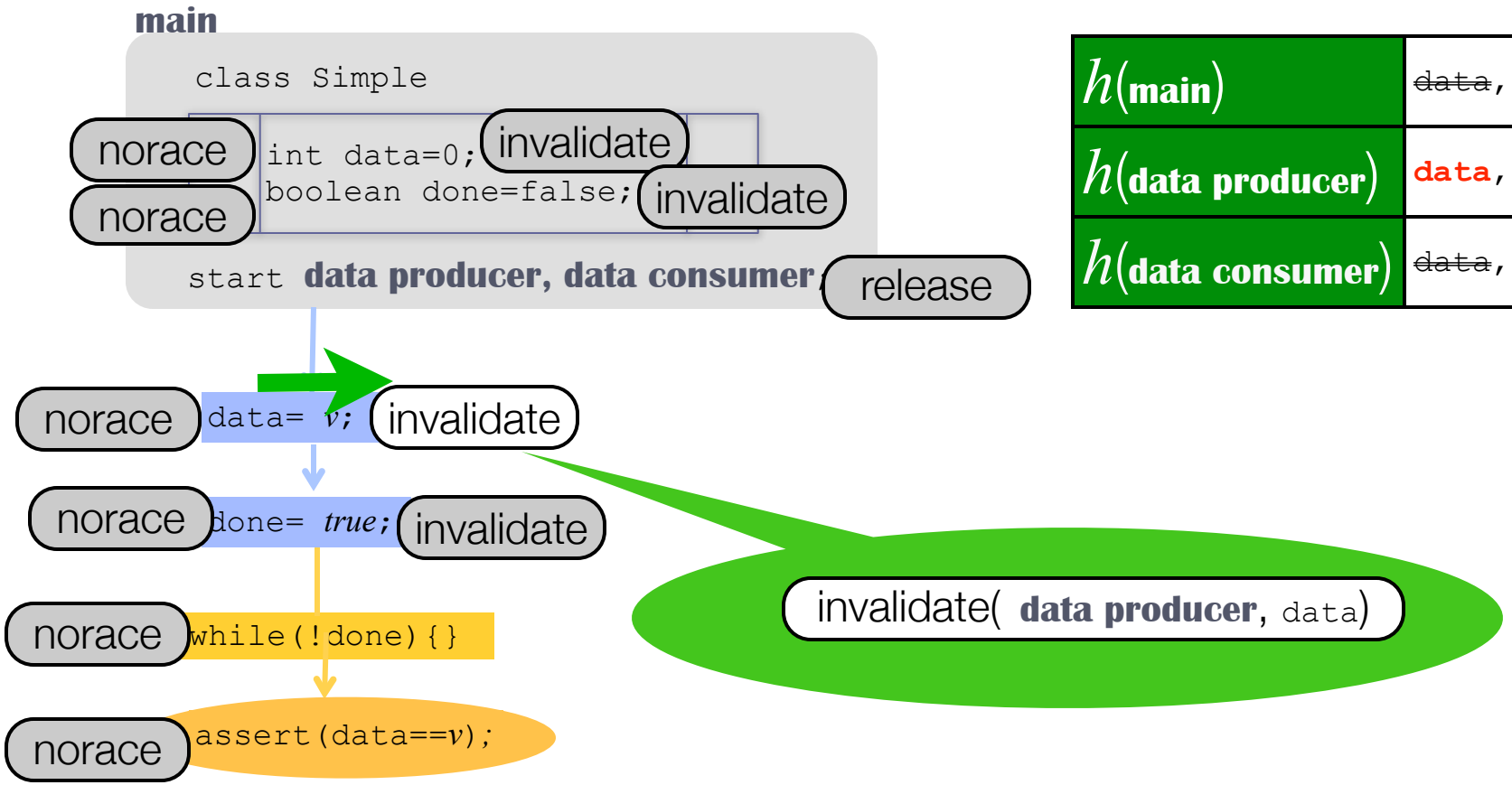


h(main)	data, done
h(data producer)	data, done
h(data consumer)	data, done

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

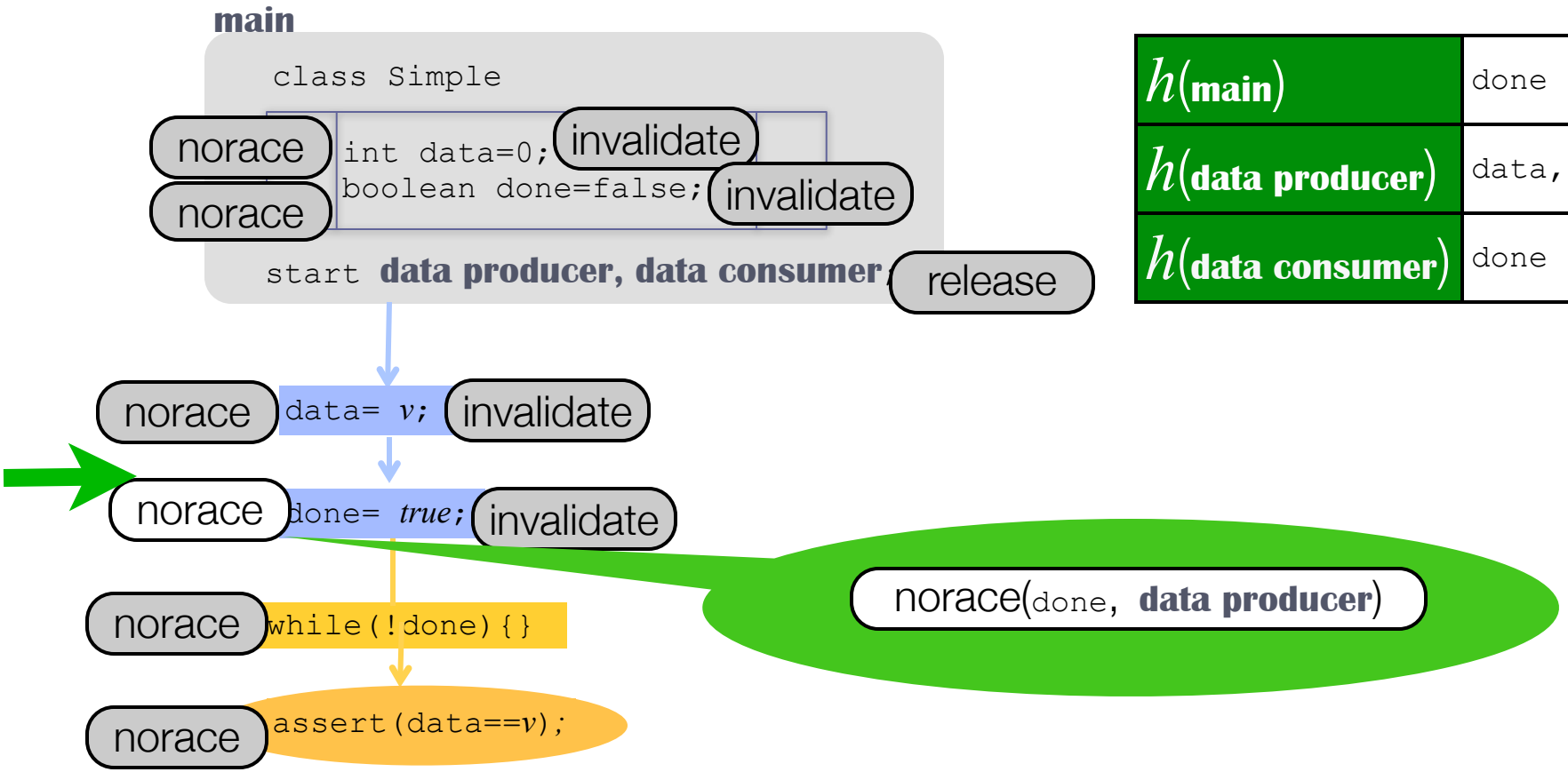


h(main)	~~data~~, done
h(data producer)	data, done
h(data consumer)	~~data~~, done

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

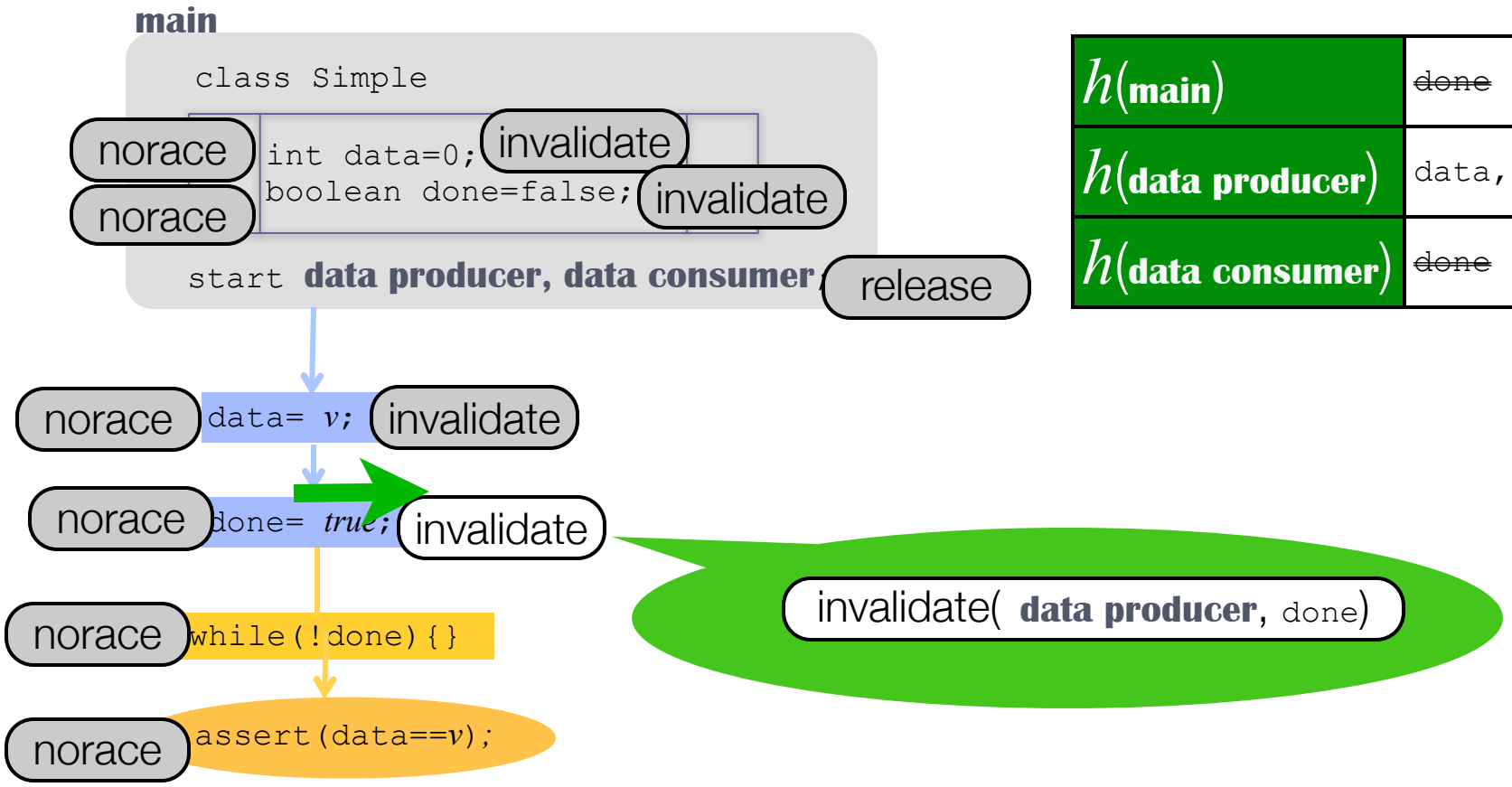


h(main)	done
h(data producer)	data, done
h(data consumer)	done

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

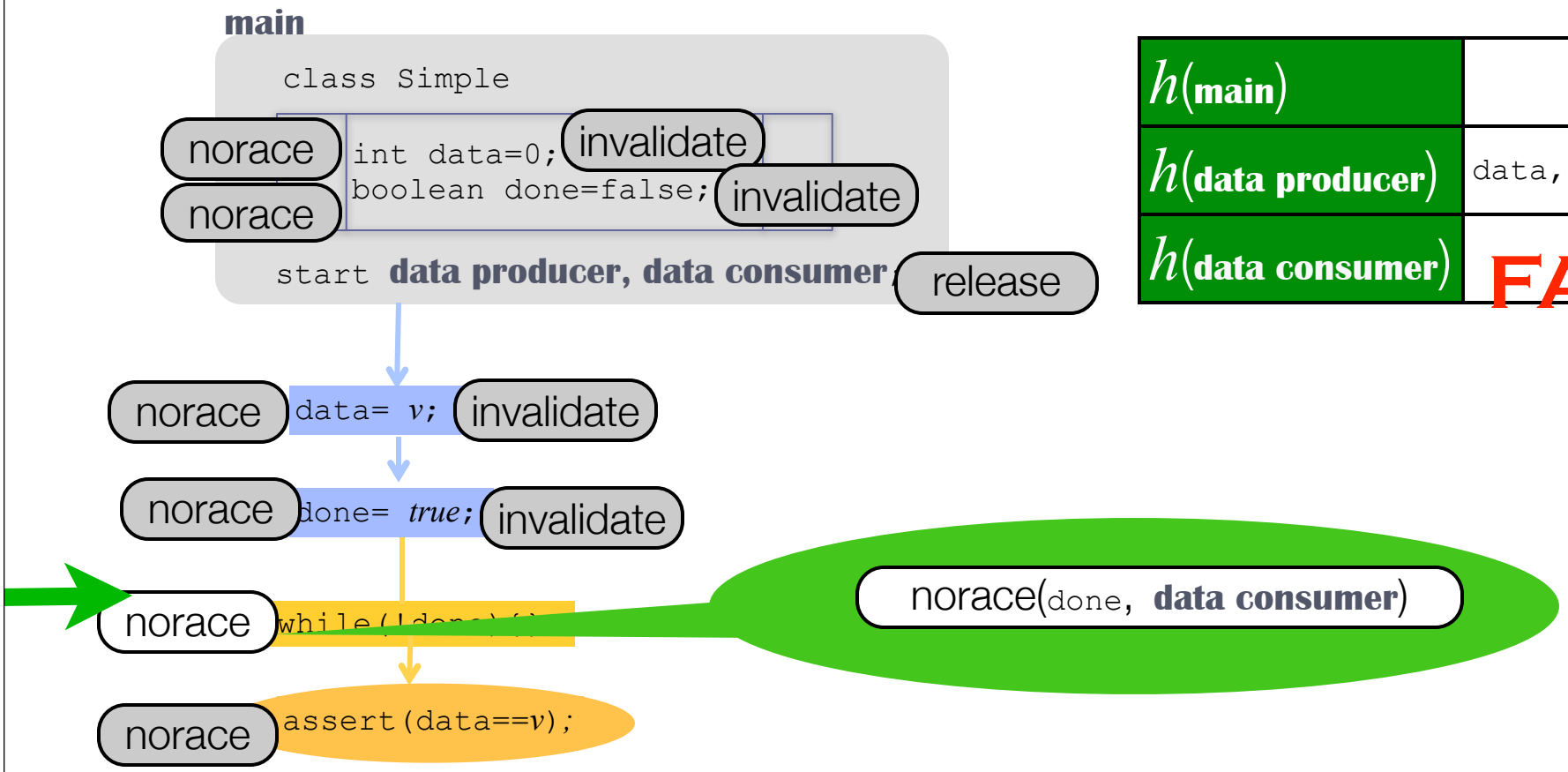


h(main)	~~done~~
h(data producer)	data, done
h(data consumer)	~~done~~

Data Race Detection

Algorithm

- Data race detection using summary function h for `Simple` java program

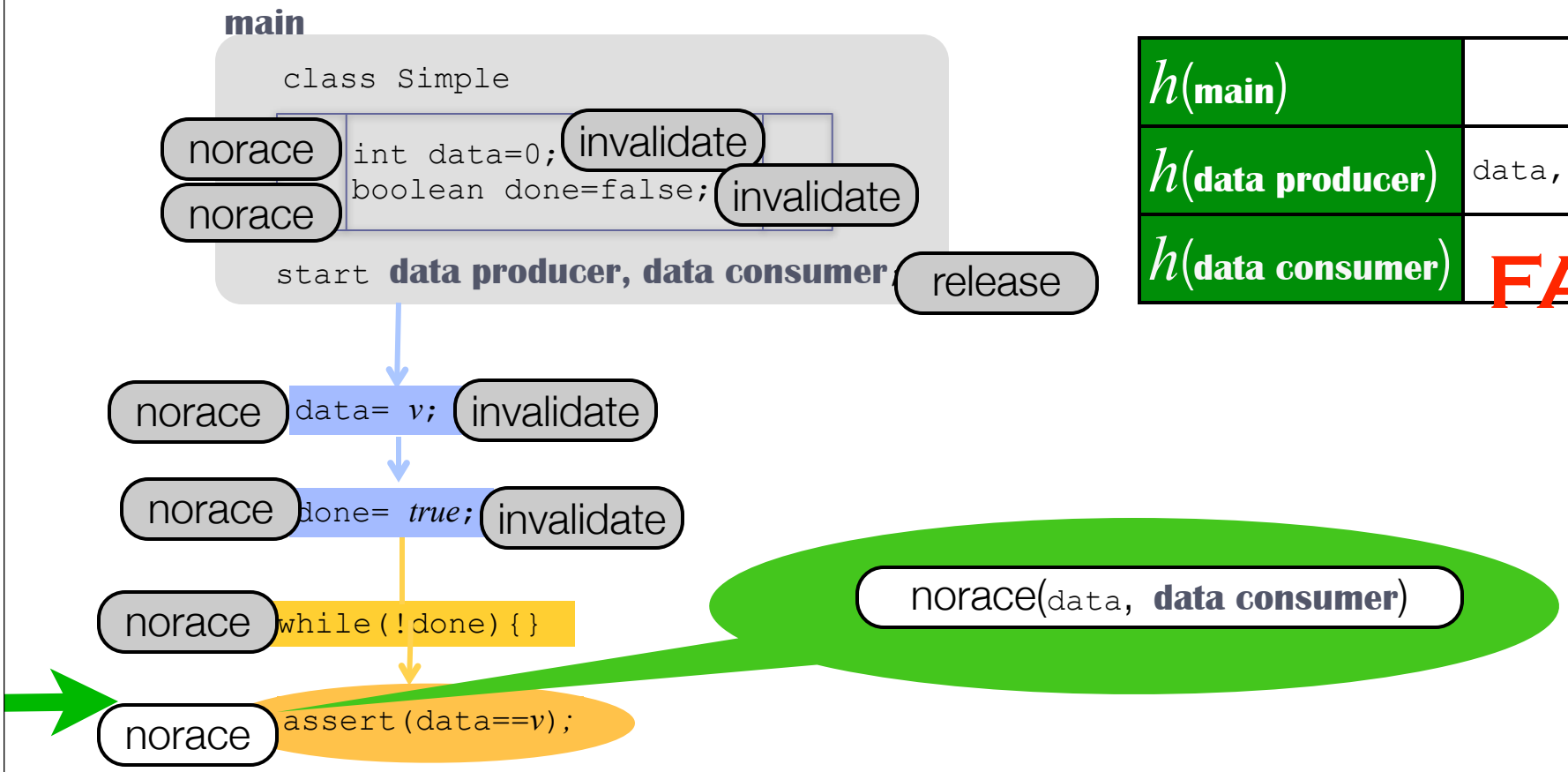


h(main)	
h(data producer)	data, done
h(data consumer)	FAIL!

Data Race Detection

Algorithm

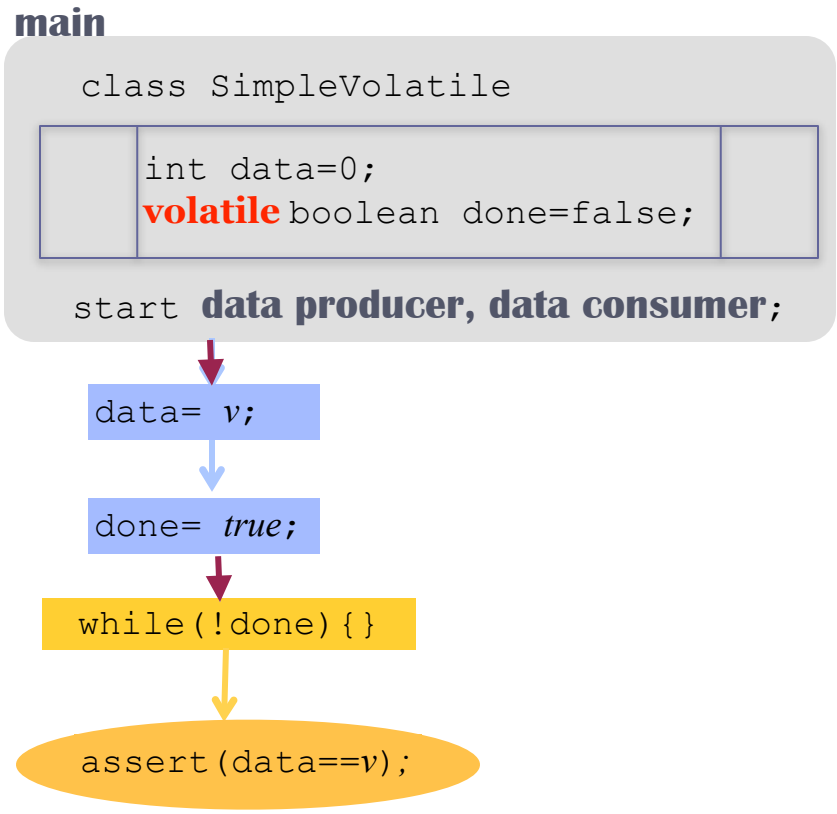
- Data race detection using summary function h for `Simple` java program



h(main)	
h(data producer)	data, done
h(data consumer)	FAIL!

Algorithm

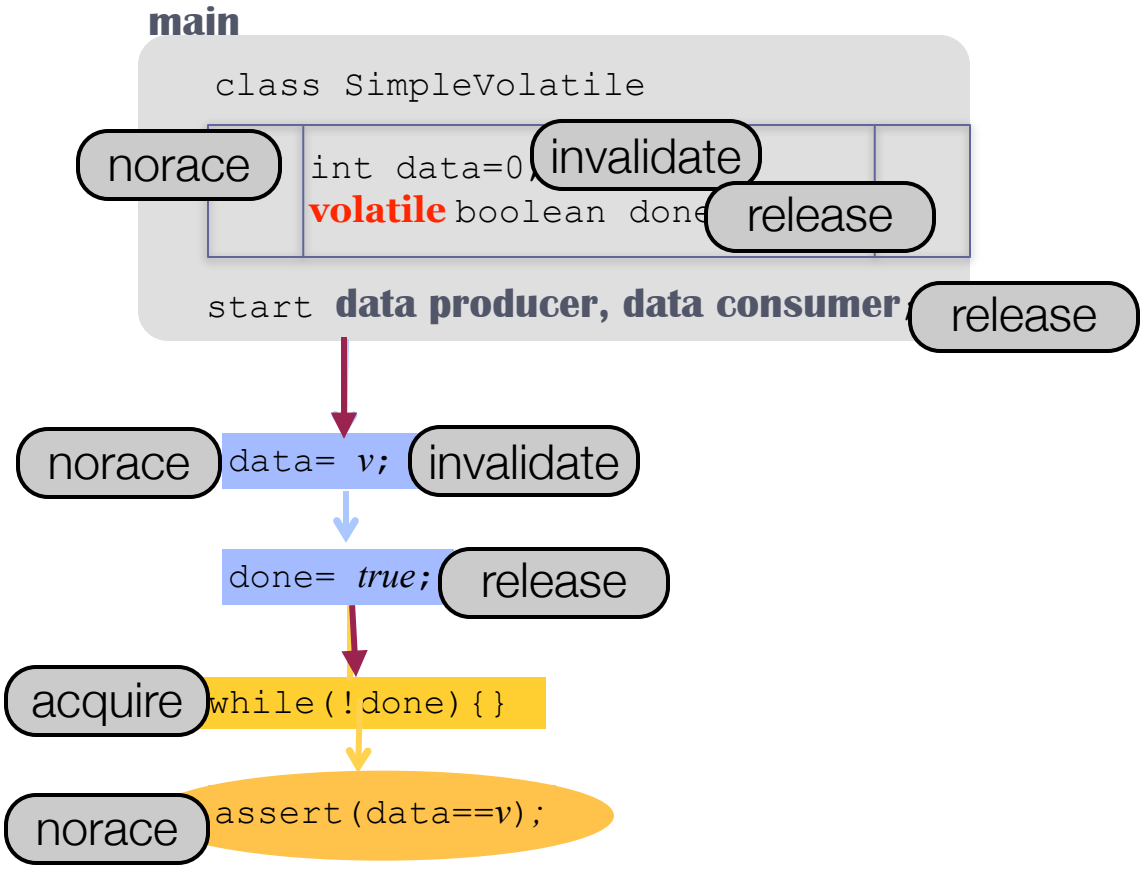
- Data race detection using summary function h for `SimpleVolatile`.



Data Race Detection

Algorithm

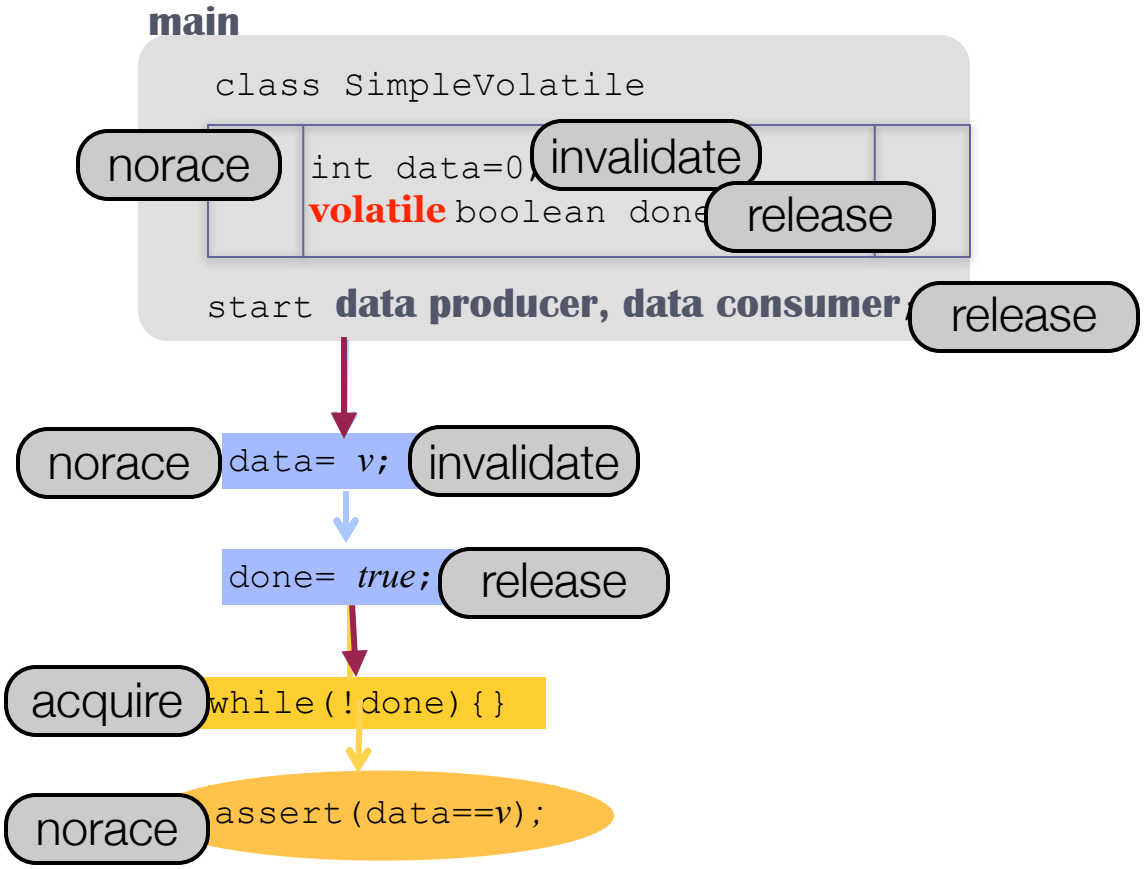
- Data race detection using summary function h for `SimpleVolatile`.



Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

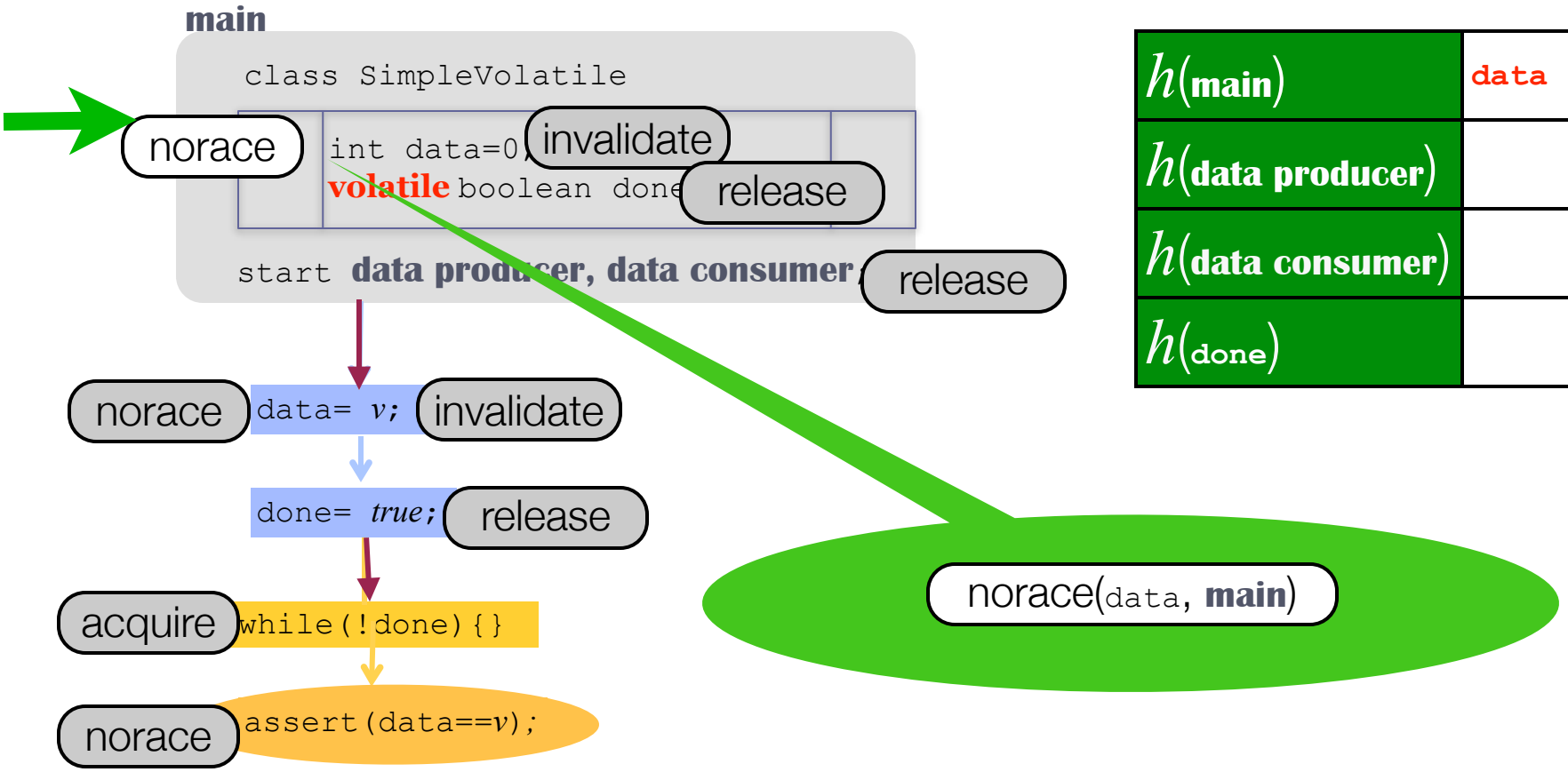


h(**main**)	`data`
h(**data producer**)	
h(**data consumer**)	
h(**done**)	

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

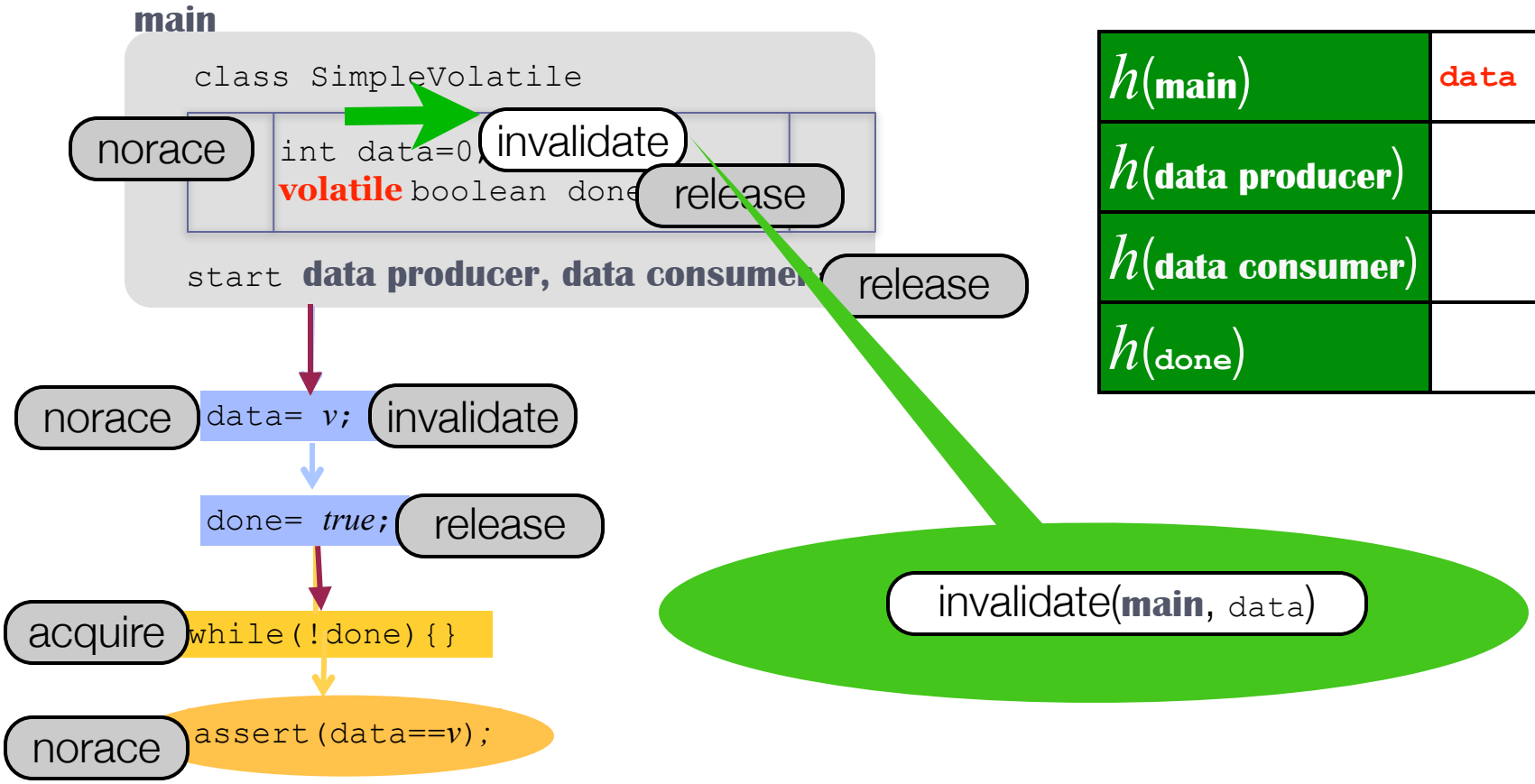


h(main)	data
h(data producer)	
h(data consumer)	
h(done)	

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

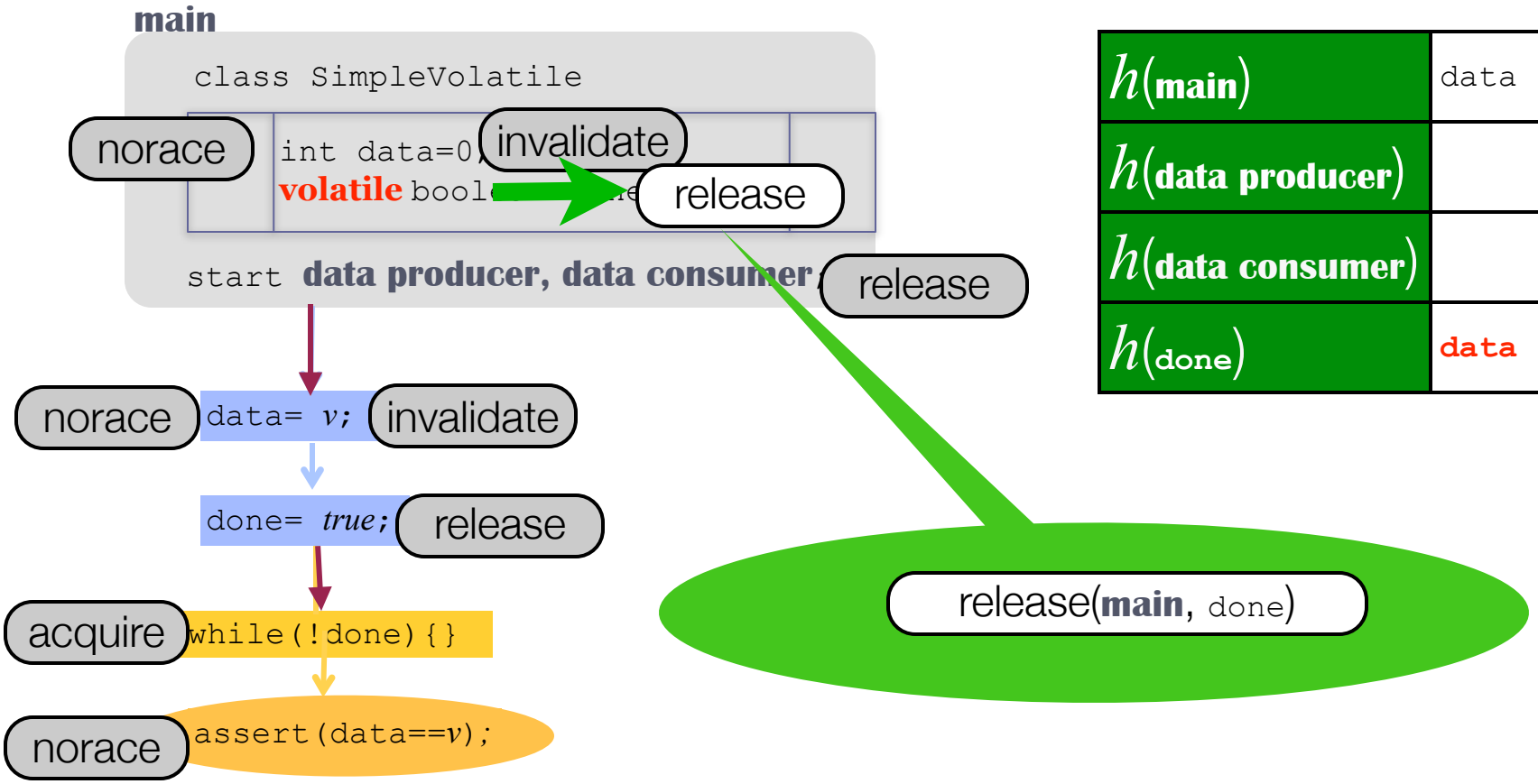


h(main)	data
h(data producer)	
h(data consumer)	
h(done)	

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

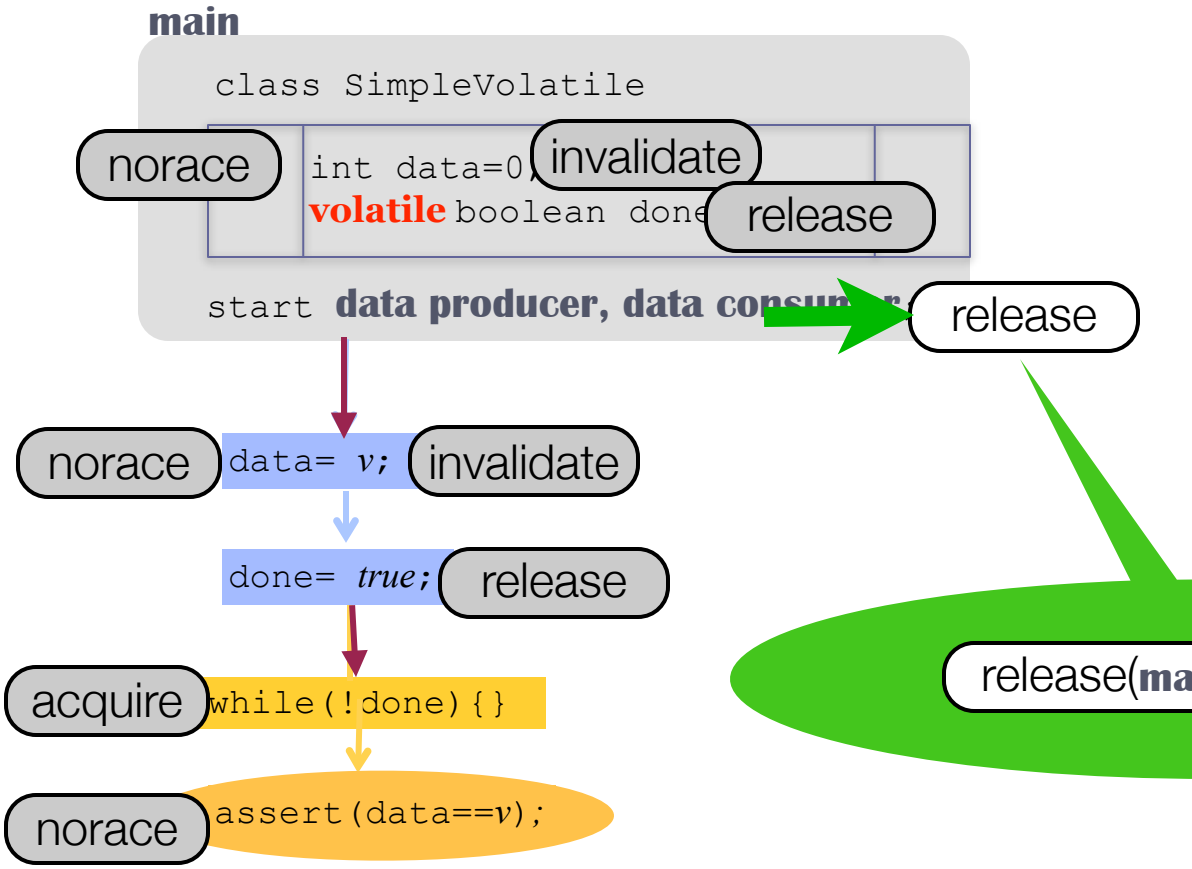


h(main)	data
h(data producer)	
h(data consumer)	
h(done)	data

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

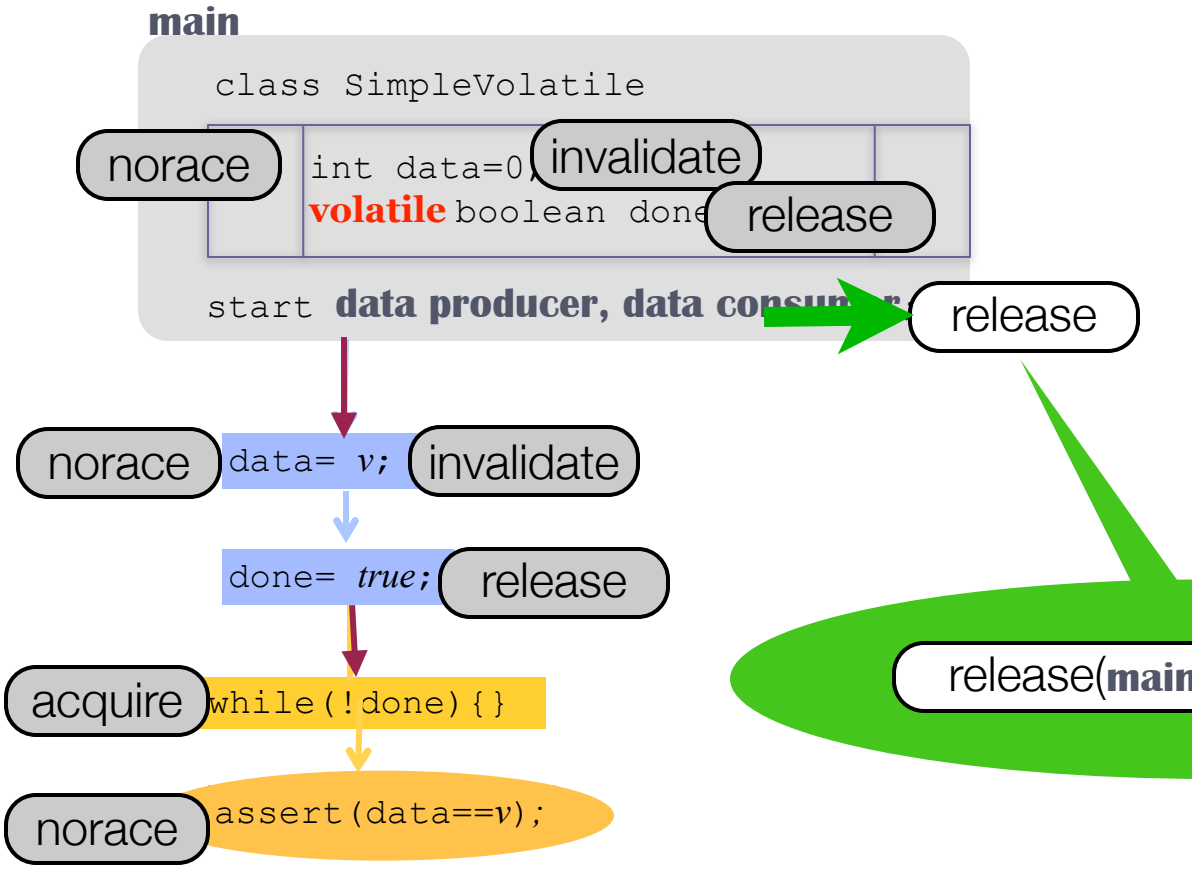


h(main)	data
h(data producer)	data
h(data consumer)	
h(done)	data

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.



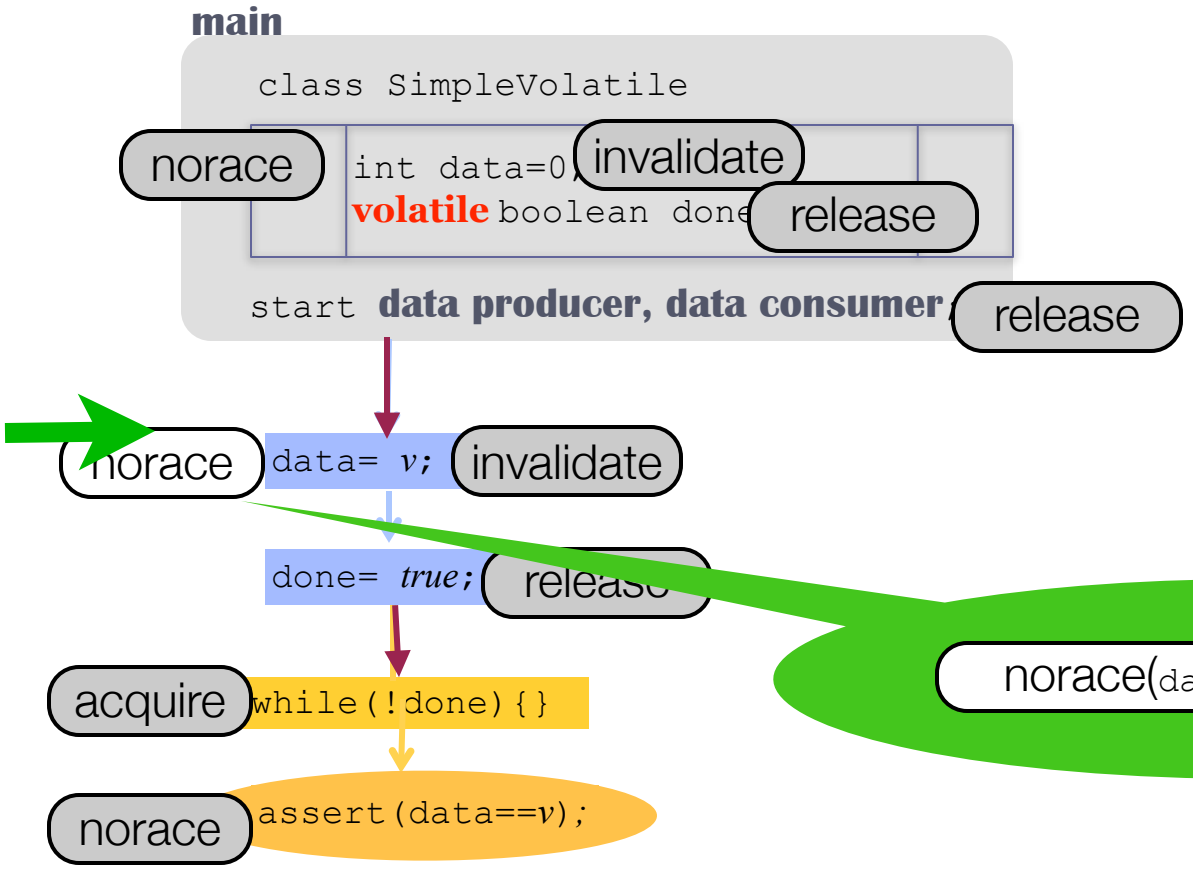
h(main)	data
h(data producer)	data
h(data consumer)	data
h(done)	data

release(main, data consumer)

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

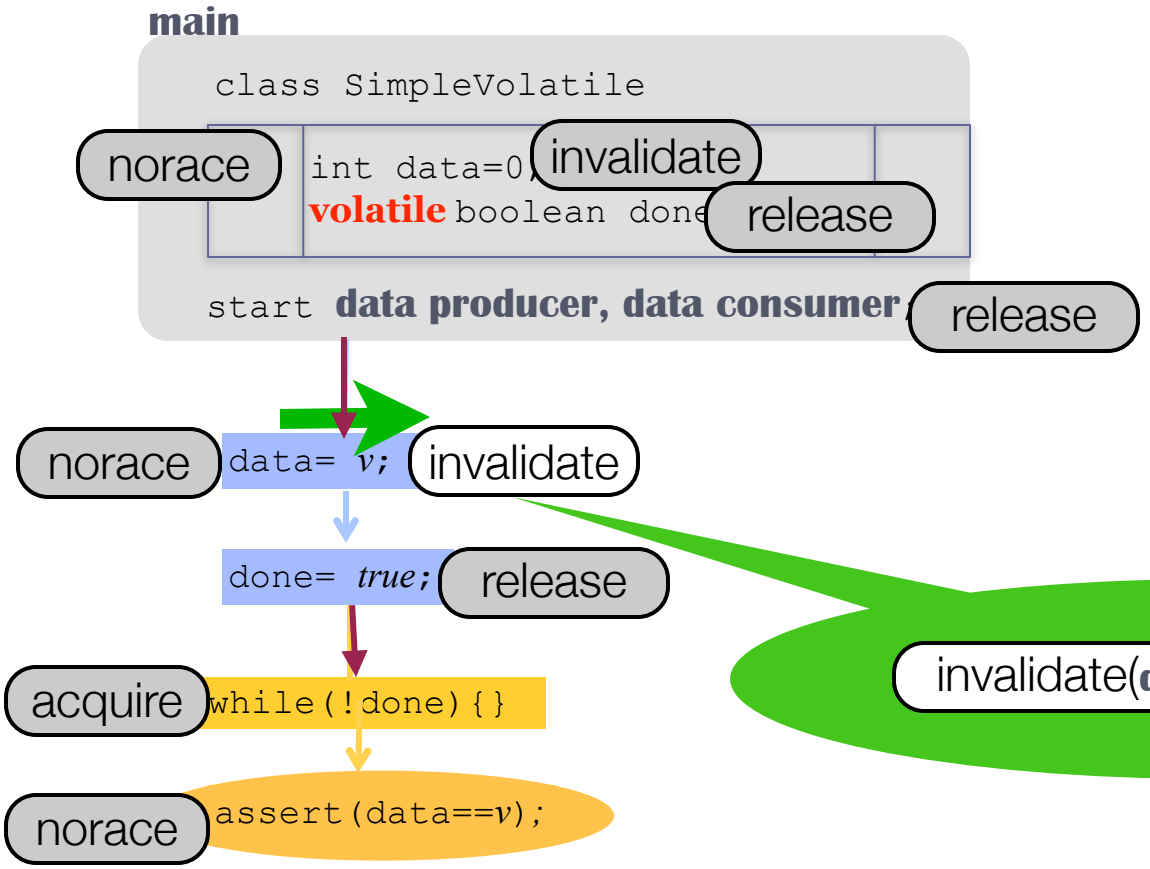


h(main)	data
h(data producer)	data
h(data consumer)	data
h(done)	data

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

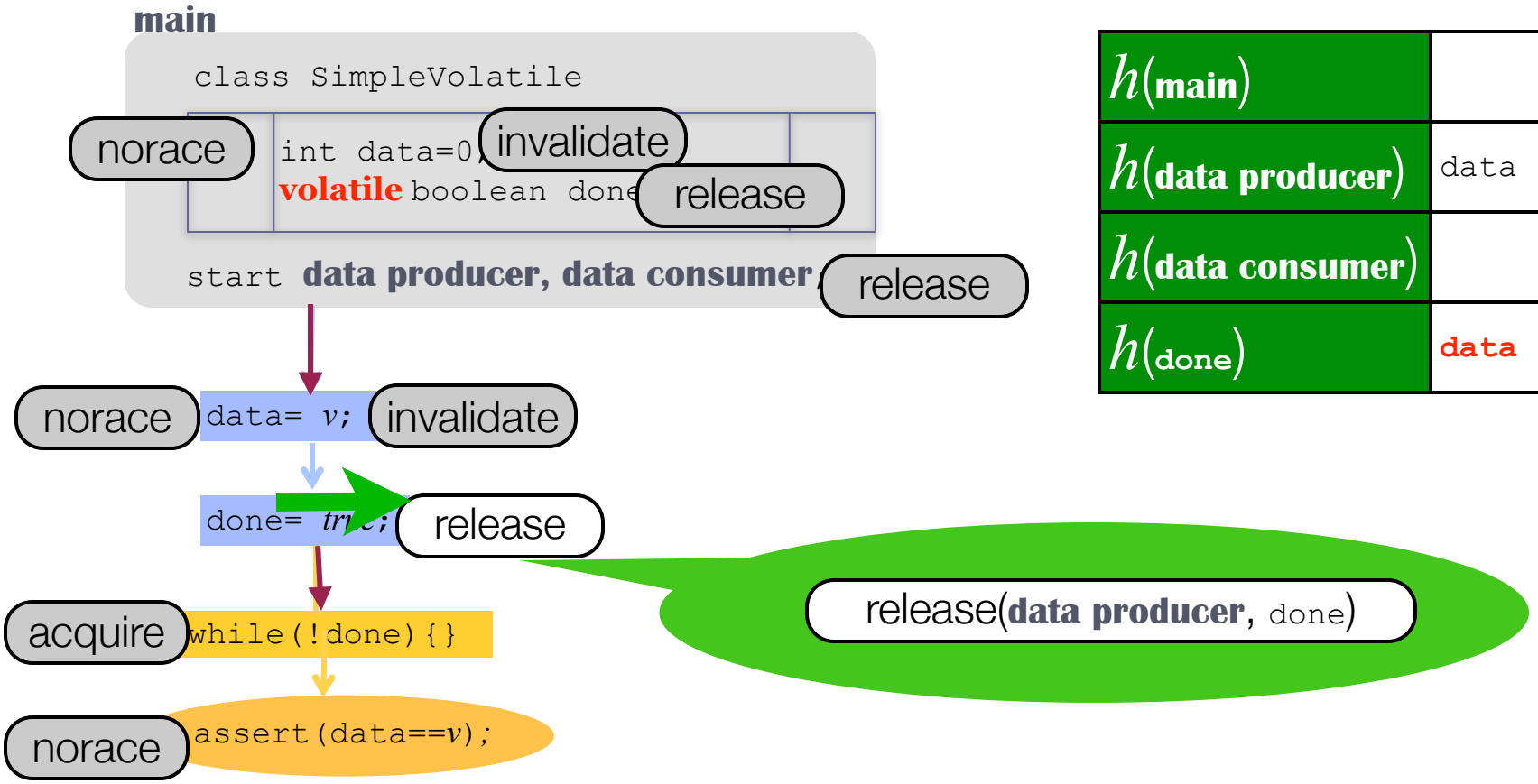


h(main)	~~data~~
h(data producer)	data
h(data consumer)	~~data~~
h(done)	~~data~~

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

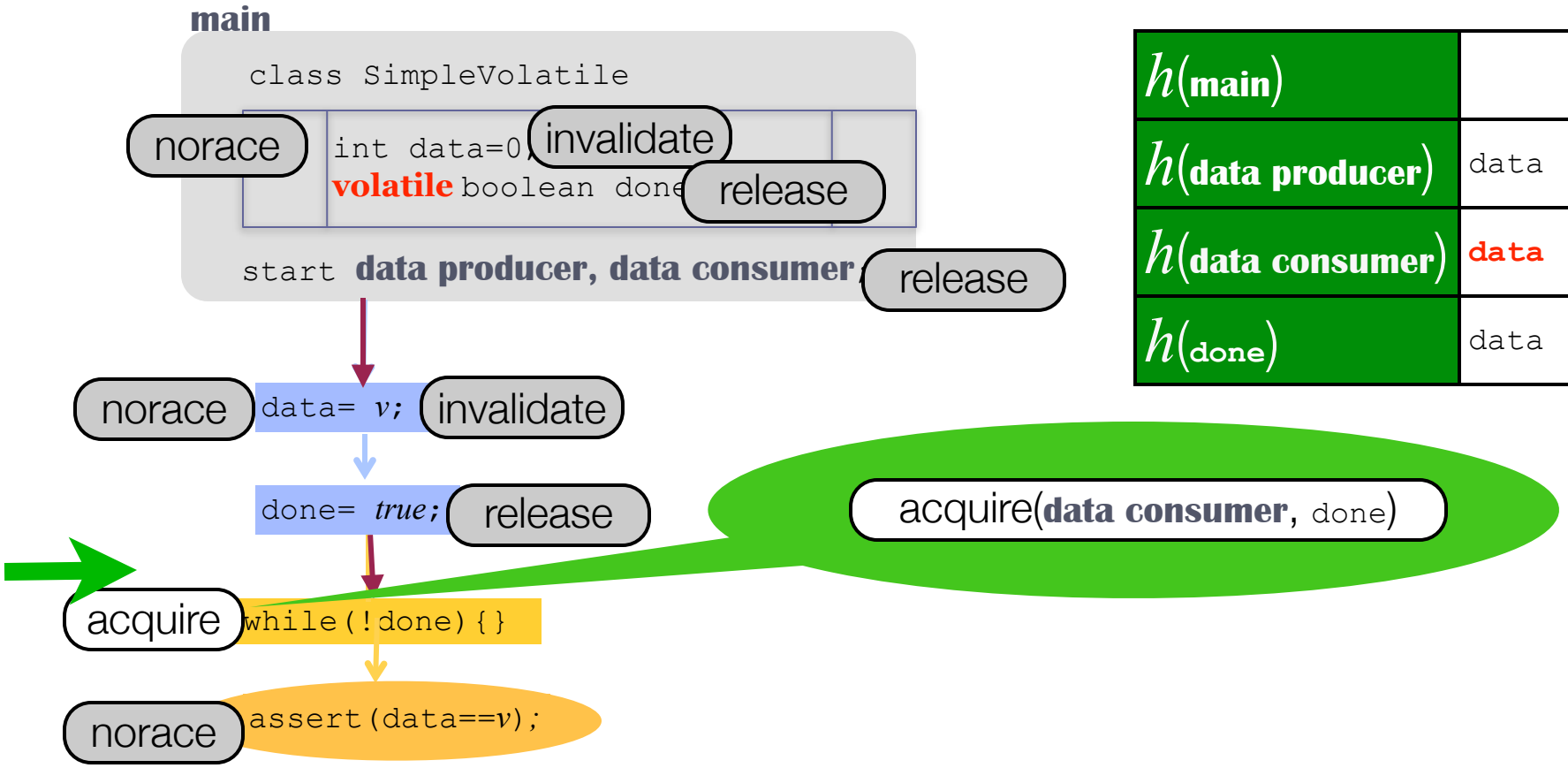


h(main)	
h(data producer)	data
h(data consumer)	
h(done)	data

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.

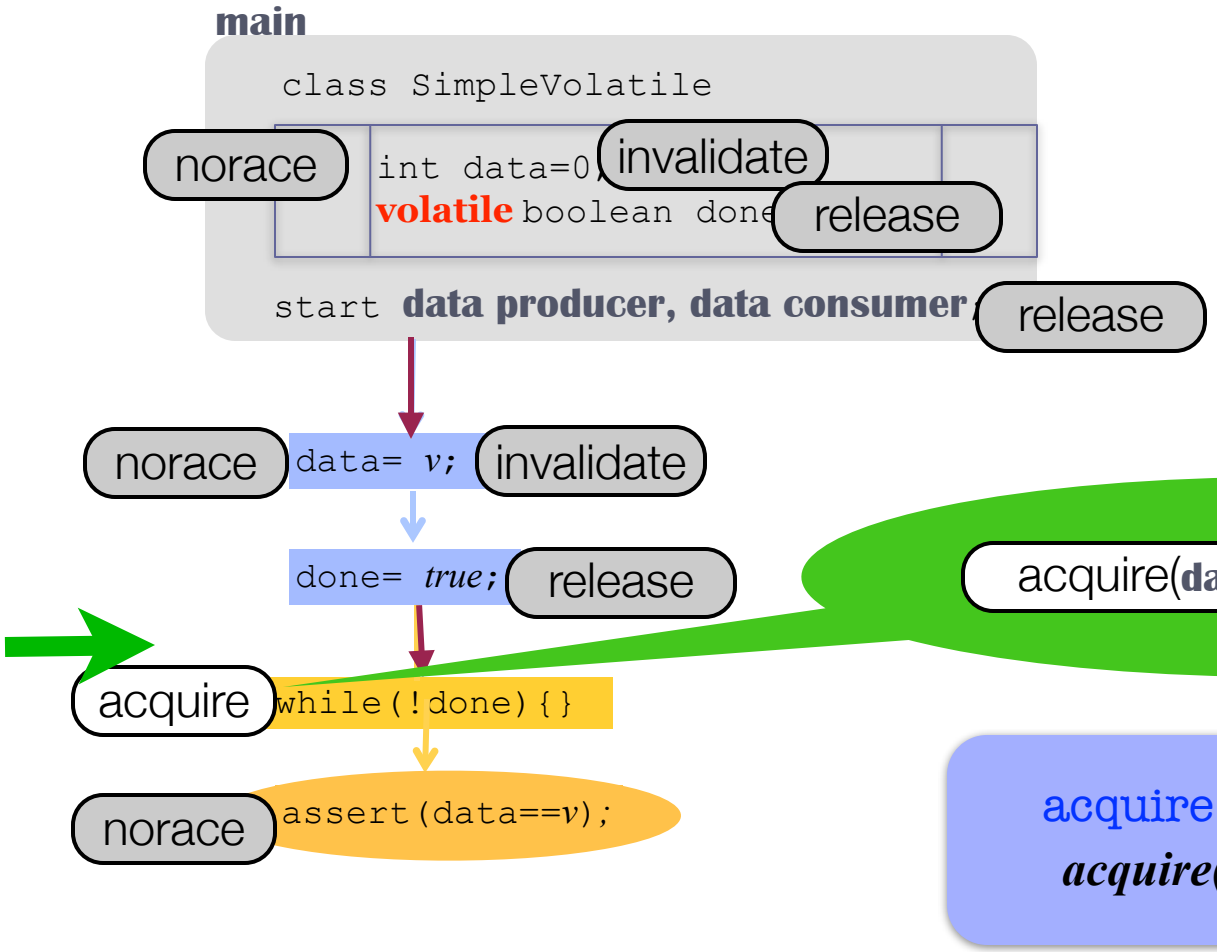


h(main)	
h(data producer)	data
h(data consumer)	data
h(done)	data

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.



h(main)	
h(data producer)	data
h(data consumer)	data
h(done)	data

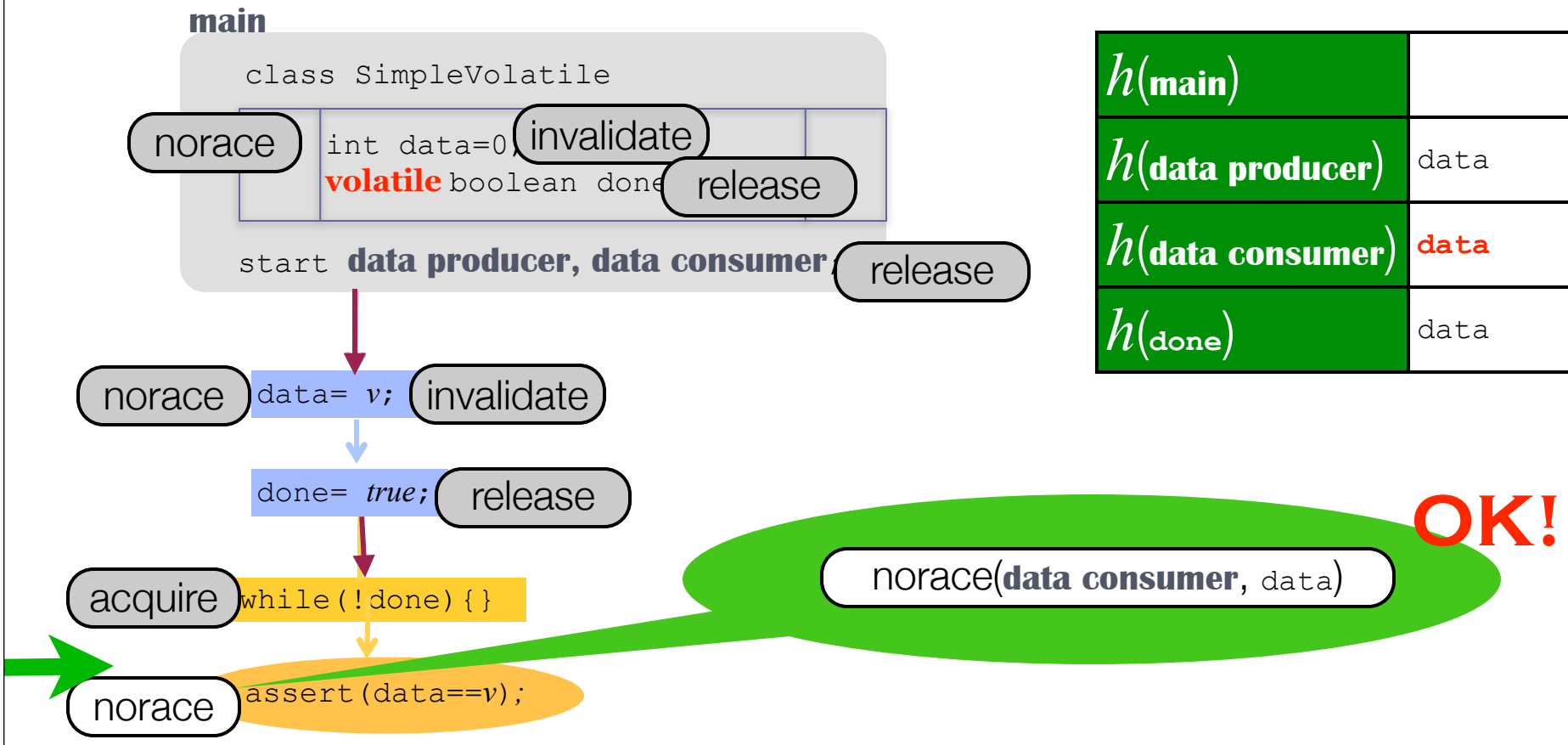
acquire adds the h(v) to h(t)

$$acquire(t, v)\ h \triangleq h[t \mapsto h(t) \cup h(v)]$$

Data Race Detection

Algorithm

- Data race detection using summary function h for `SimpleVolatile`.



h(main)	
h(data producer)	data
h(data consumer)	data
h(done)	data

Algorithm

- Summary Function h

$$h : SynchAddr \cup Threads \rightarrow 2^{Addr}$$

1. Initially, $h_0 = \lambda z.$if $z{=}main$ then $static(P)$ else $\perp$
2. $release(t, x)\ h \triangleq h[x \mapsto h(t) \cup h(x)]$
3. $acquire(t, x)\ h \triangleq h[t \mapsto h(t) \cup h(x)]$
4. $invalidate(t, x)\ h \triangleq \lambda z.$if $(t{=}z)$ then $h(z)$ else $h(z) \backslash \{x\}$
5. $new(t, fields, volatiles)\ h \triangleq \lambda z.$if $(t{=}z)$ then $h(t) \cup fields$
   else if $(z \in volatiles)$ then $\{\}$
   else $h(z)$
6. $norace(x, t) = x \in h(t)$

new instantiates h for volatiles and add fields to h(t)

Experimental Result

Data Race Analysis

Experimental Result

- Case Studies

	number of tests with races	total tests
goldilock model checking examples	0	7
Herily-Shavit concurrent data structure examples	19	68
Amino concurrent basic blocks	4	30
Google concurrent ADT experiment framework	0	4
Google concurrent data structures workshop barrier	10	12
Java Grand Forum benchmark	6	11
Total	39	132

Experimental Result

- Types of races
  - ➡ missing synchronization
  - ➡ unsafe publication
  - ➡ using volatile array instead of *java.util.concurrent.atomic* package
  - ➡ missing volatile declaration
  - ➡ benign race

Experimental Result

- Unsafe Publication

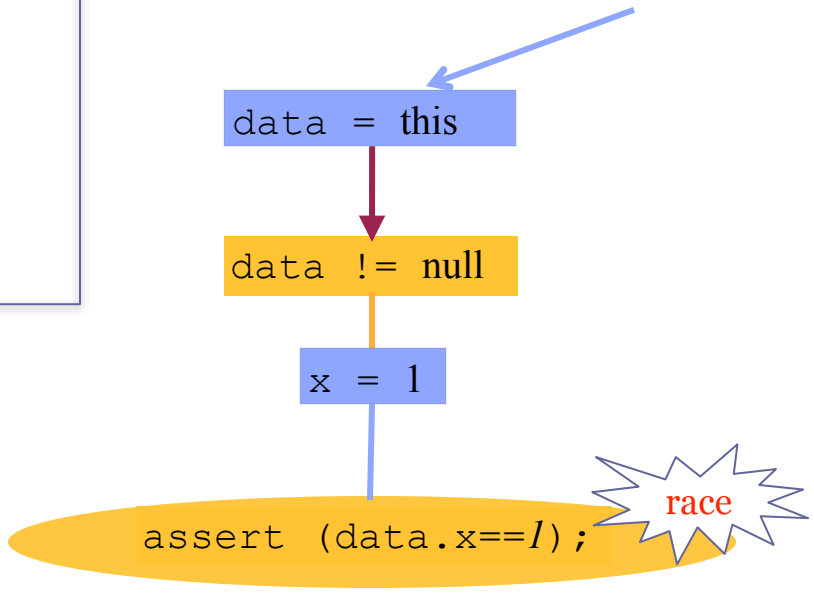
```
static volatile Data data;
static class Data {
  int x;
   Data()
   {
      data = this;
      x = 1;
   }
```

Thread1

```
new Data();
```

Thread2

```
if (data != null)
  assert(data.x==1);
```



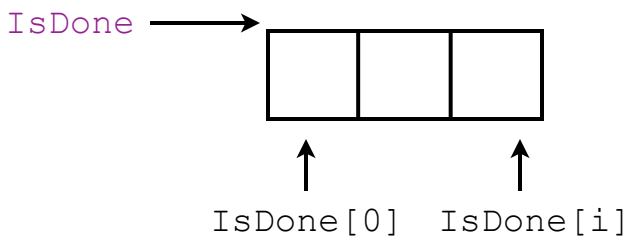
Experimental Result

- Types of races
  - ➡ missing synchronization
  - ➡ unsafe publication
  - ➡ using volatile array instead of *java.util.concurrent.atomic* package
  - ➡ missing volatile declaration
  - ➡ benign race

Experimental Result

- Using volatile array

```
volatile boolean[] IsDone;
 . . .
IsDone[id] = ...

while (!IsDone[i])...
```



Conclusion

Conclusion

- JRF found races from concurrent data structures which implements lock-free or wait-free algorithms.

  ➡ those cannot be handled properly in existing race detectors which uses lock-based algorithm

  ➡ those libraries are implemented by expert concurrent programmers but still have races

- JRF also found races from java standard library classes

- After applying JRF, the program verified to be a race free can soundly use JPF to check further properties.

Future Works

- JRF-E (JRF-Eliminator)

  : suggests how to eliminate found races

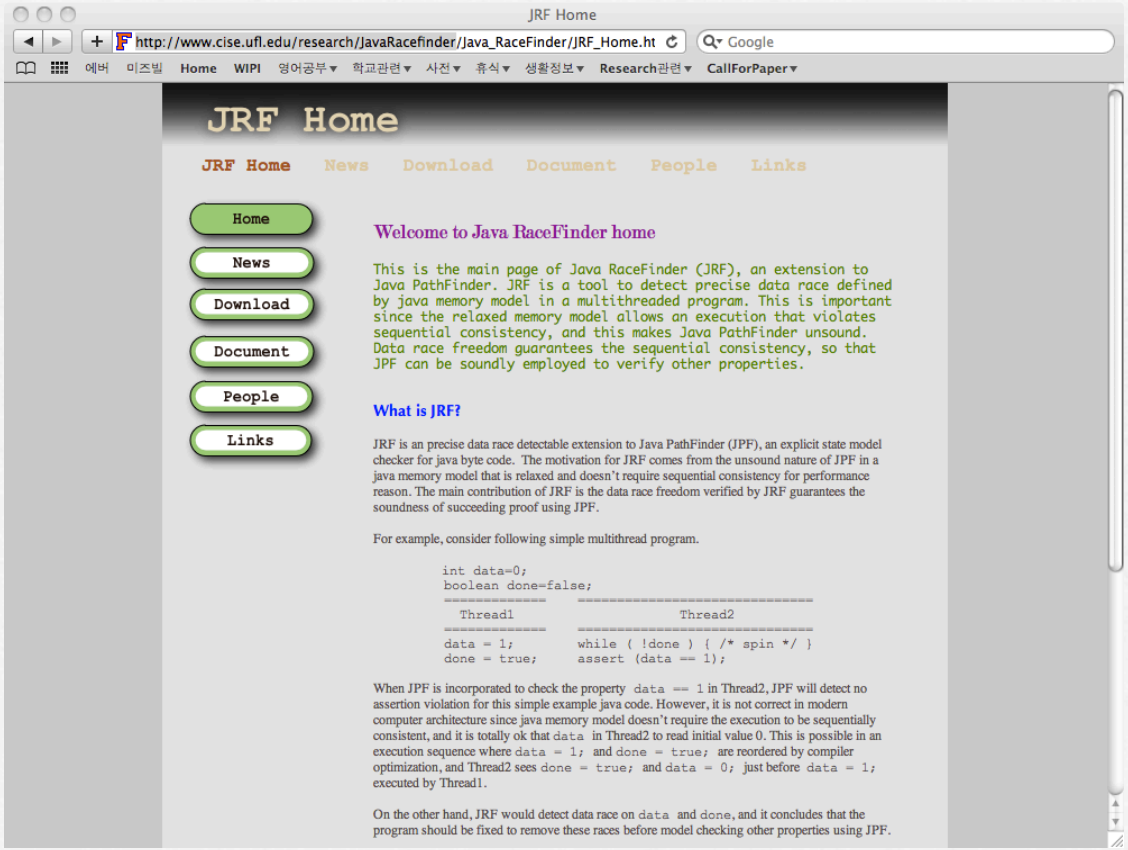
- JRF-M (JRF-Modular)

  : modular race detection using symbolic execution

Thank you!

JRF version1.0

- **http://www.cise.ufl.edu/research/JavaRacefinder/**



Publications

Paper

Precise data race detection in a relaxed memory model using heuristic-based model checking
Kim, K., Yavuz-Kahveci, T., and Sanders, B. A. 2009a.. In Proceedings of the 24th ACM/IEEE Conference on Automated Software Engineering

Assertional reasoning about data races in a relaxed memory models
Sanders, B. A. and Kim, K. 2008. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming

Precise data race detection in a relaxed memory model using heuristic-based model checking
Kim, K., Yavuz-Kahveci, T., and Sanders, B. A. submitted for publication

Detecting and eliminating data races using counterexample and acquiring his tory analysis
Kim, K., Yavuz-Kahveci, T., and Sanders, B. A. in preparation

Technical Report

Precise data race detection in a relaxed memory model using model checking
Kim, K., Yavuz-Kahveci, T., and Sanders, B. A. 2009b.. Tech. Rep. REP-2009-480, University of Florida.

Extending JPF

- JPF PreciseRaceDetector.

  (1) failed to detect a race on volatile array which is accessed with interval
  (2) found a false race on volatile field
  (3) would not detect races involved in MJI code
  (4) cannot handle Unsafe publication
  (5) JRF can provide suggestions based on counterexample analysis and acquiring history

Extending JPF

- JPF PreciseRaceDetector.

  (1) failed to detect a race on volatile array which is accessed with interval
  (2) found a false race on volatile field
  (3) would not detect races involved in MJI code
  (4) cannot handle Unsafe publication
  (5) JRF can provide suggestions based on counterexample analysis and acquiring history

	False race on volatile	Missed race on volatile array element
Herily-Shavit (19)	1	5
Google (10)	7	2
Amino (4)	0	0
JGF (6)	0	4

Extending JPF

- Unsafe Publication

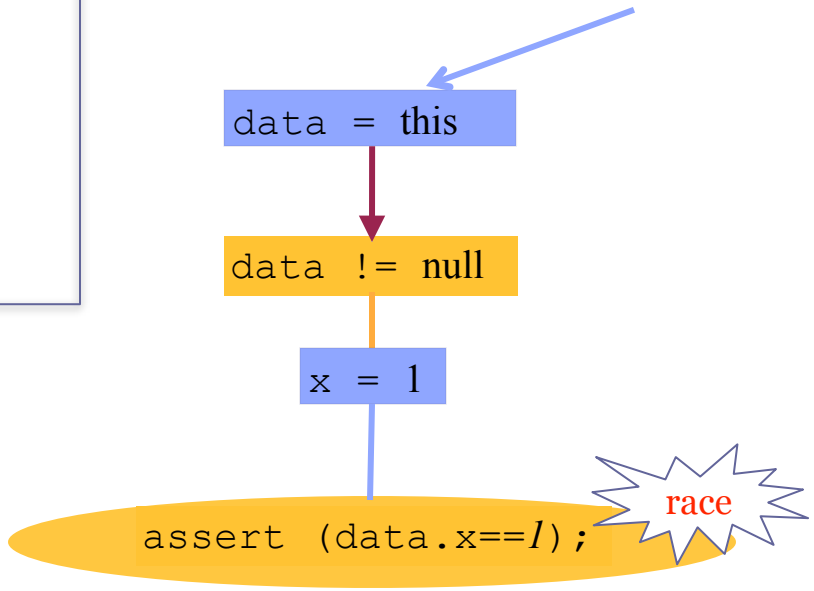
```
static volatile Data data;
static class Data {
  int x;
   Data()
   {
      data = this;
      x = 1;
   }
```

Thread1

```
new Data();
```

Thread2

```
if (data != null)
  assert(data.x==1);
```



Extending JPF

- Unsafe Publication

```
static volatile Data data;
static class Data {
  int x;
   Data()
   {
      data = this;
      x = 1;
   }
```

Thread1

```
new Data();
```

Thread2

```
if (data != null)
  assert(data.x==1);
```