

# 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**

```
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)”.

```
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)”.

class

We can check this property  
by exploring **all possible interleavings**  
of two threads

## **data producer**

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

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

# Motivating Example

---

data	0
done	false

## data producer

```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/ }
```

```
assert (data==v) ; /*consume*/
```

```
data = v;
```

```
while (!done) { }
```

# Motivating Example

---

data	<i>v</i>	
done	<i>false</i>	

## data producer

```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/ }
```

```
assert (data==v) ; /*consume*/
```

```
data = v;
```

```
while (!done) { }
```

# Motivating Example

data	<i>v</i>
done	<i>false</i>

## data producer

```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/ }
```

```
assert (data==v) ; /*consume*/
```

```
data = v;
```

```
done = true;
```

```
while (!done) { }
```

```
while (!done) { }
```



# Motivating Example

data	<i>v</i>	
done	<i>true</i>	

## data producer

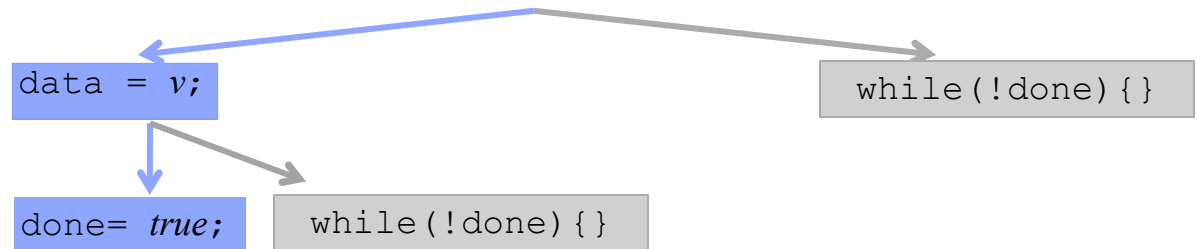
```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/ }
```

```
assert (data==v) ; /*consume*/
```



# Motivating Example

data	<i>v</i>
done	<i>true</i>

## data producer

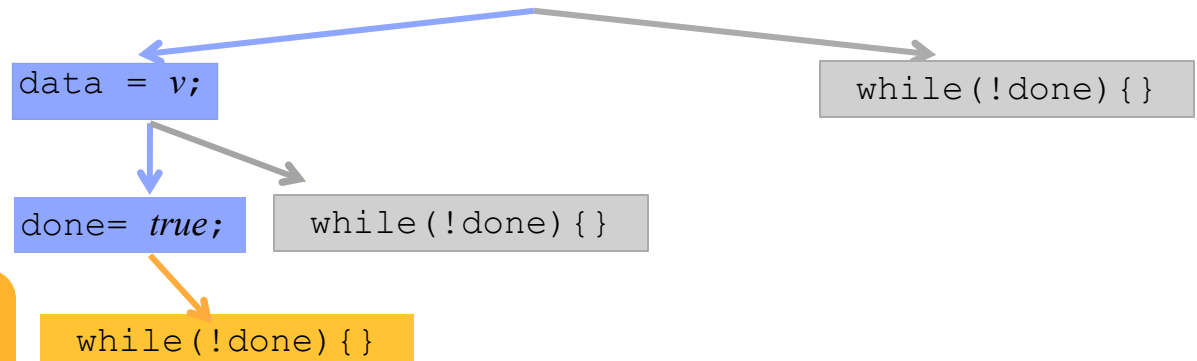
```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/ }
```

```
assert (data==v) ; /*consume*/
```



# Motivating Example

data	<i>v</i>
done	<i>true</i>

## data producer

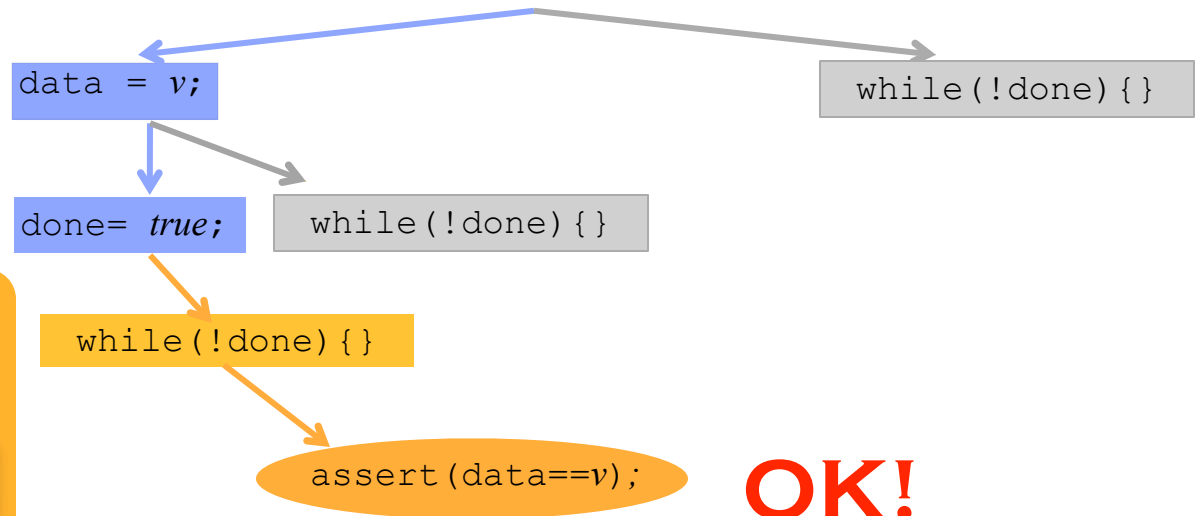
```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/
```

```
assert (data==v); /*consume*/
```



# Motivating Example

data	<i>v</i>
done	<i>true</i>

## data producer

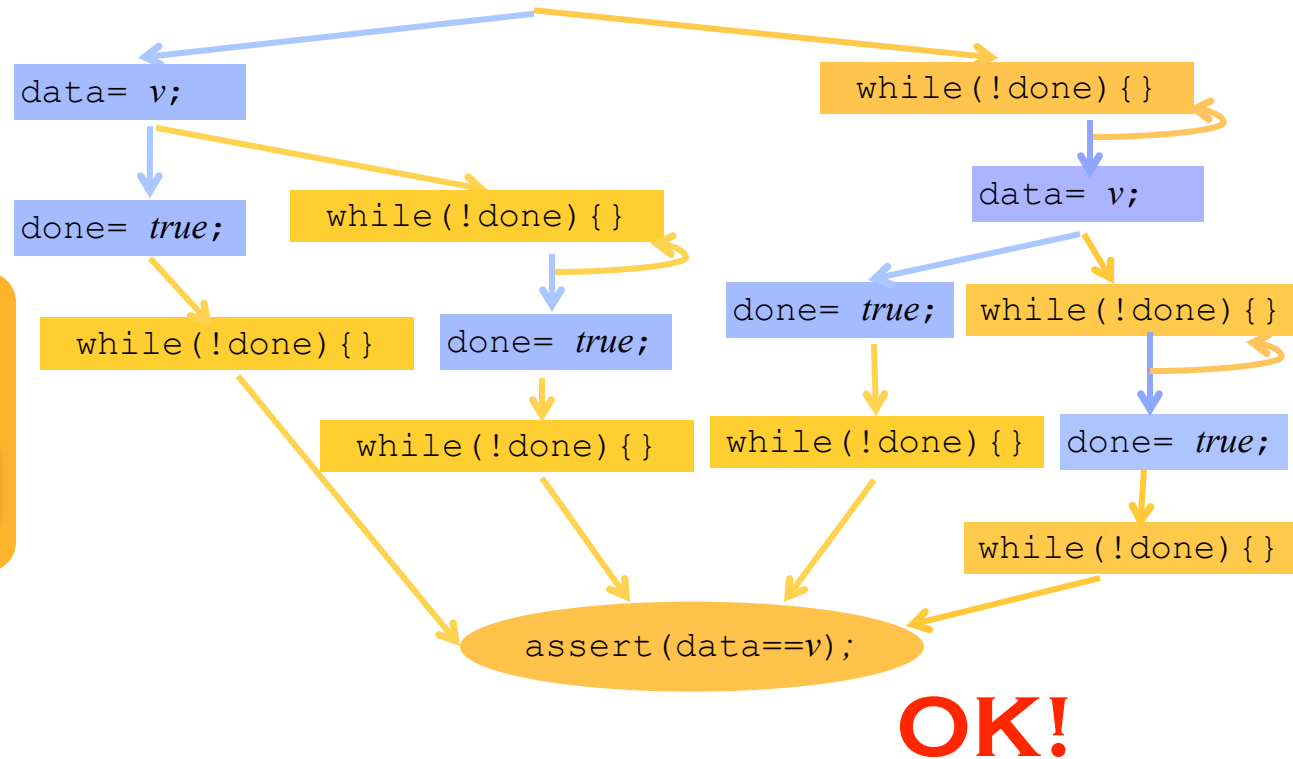
```
data = v; /*produce*/
```

```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/
```

```
assert (data==v); /*consume*/
```



# Motivating Example

data	<i>v</i>
done	<i>true</i>

## data producer

```
data = v; /*produce*/
```

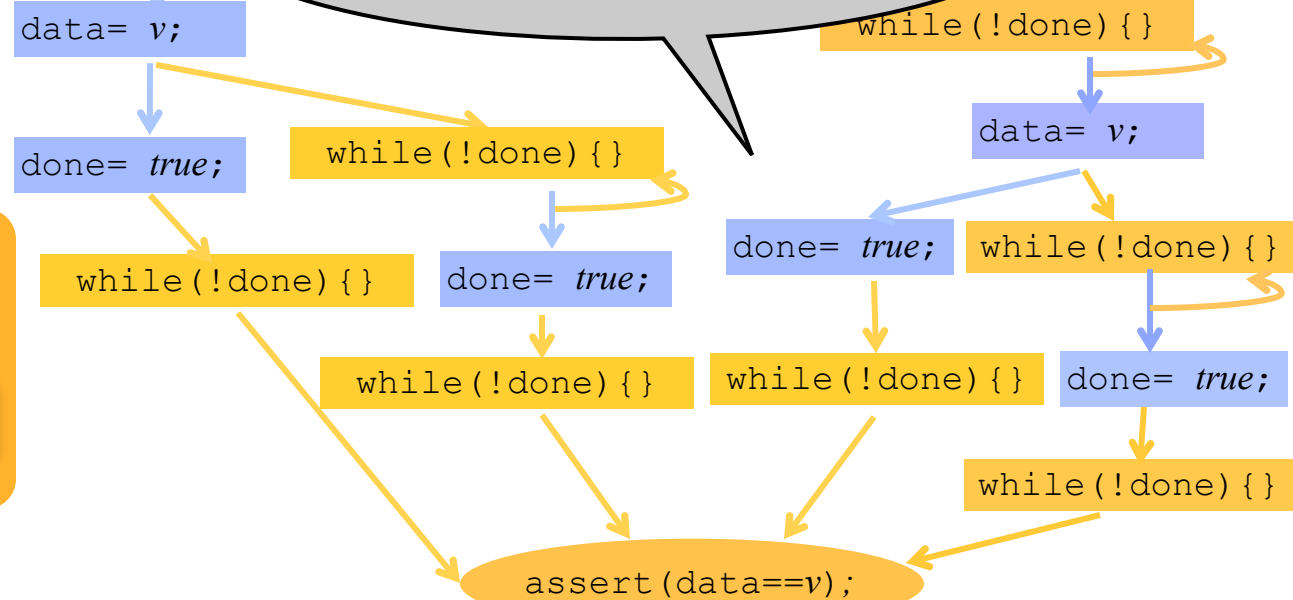
```
done = true; /*notify*/
```

## data consumer

```
while (!done) { /*spin*/
```

```
assert (data==v); /*consume*/
```

Model checking




OK!

# Motivating Example

---

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



Java PathFinder  
result

```
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**

```
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**

## Data Race

When two memory accesses by different threads on the same memory location are not ordered, and at least one is a write

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

```
assert (data==v) ; /*consume*/
```



# 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**

```
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*/
```

data race on data

# 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**

```
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*/
```

data race on done

# Motivating Example

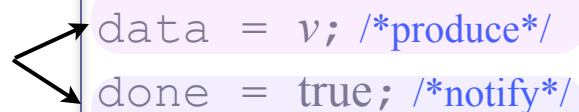
---

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

```
class Simple
```

	<pre>int data=0; boolean done=false;</pre>	
--	--	--

**data producer**

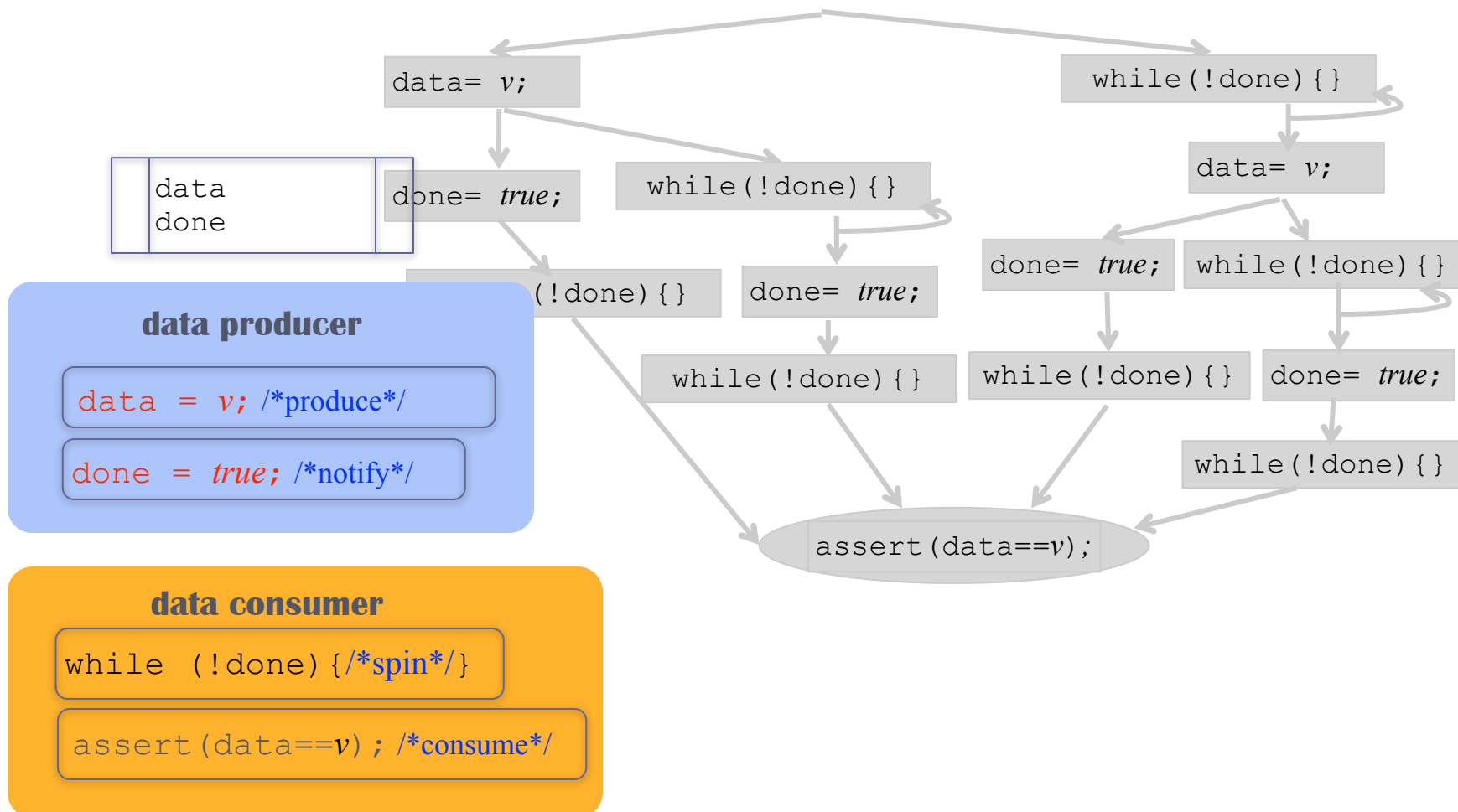


```
data = v; /*produce*/  
done = true; /*notify*/
```

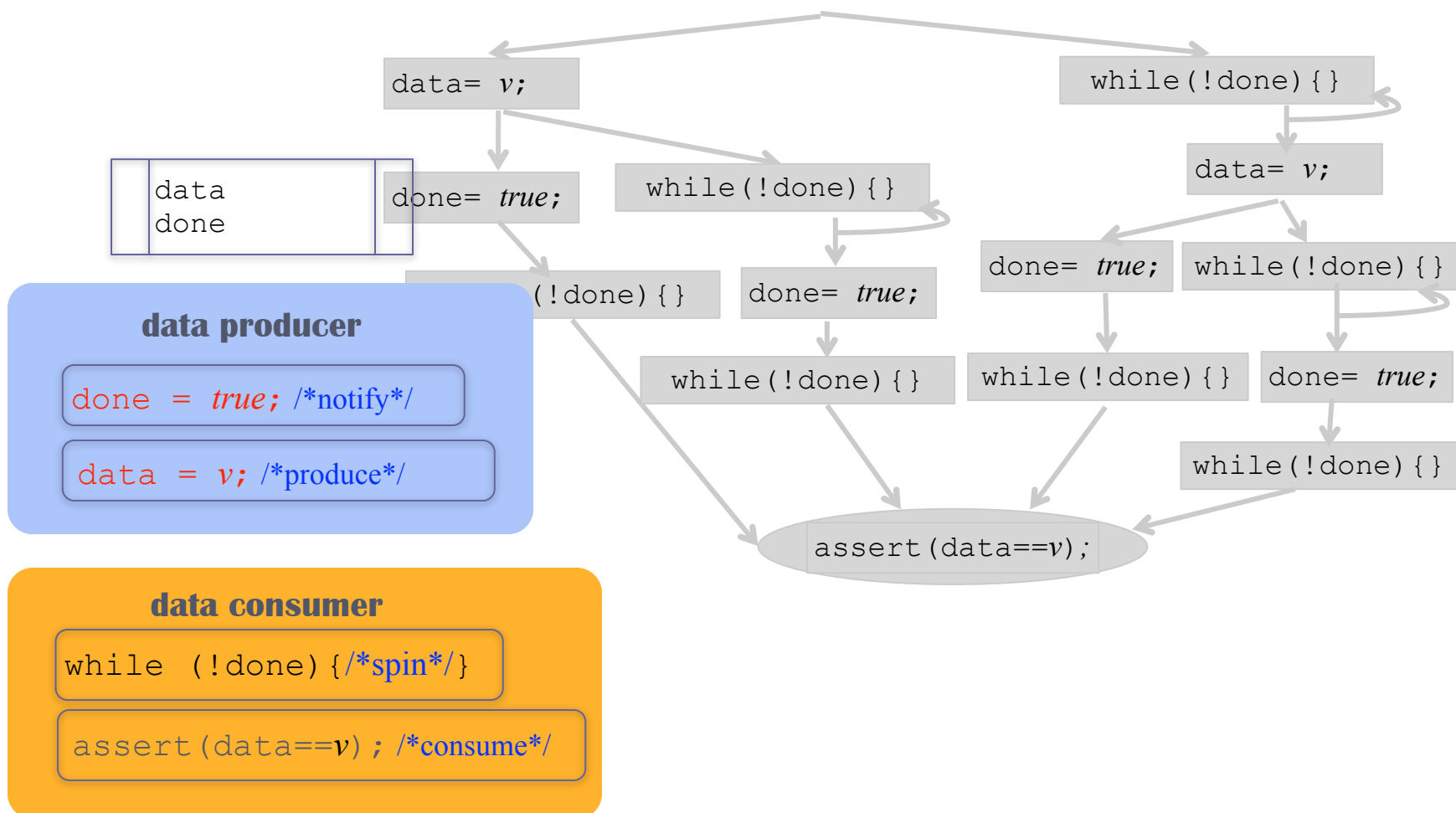
**data consumer**

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

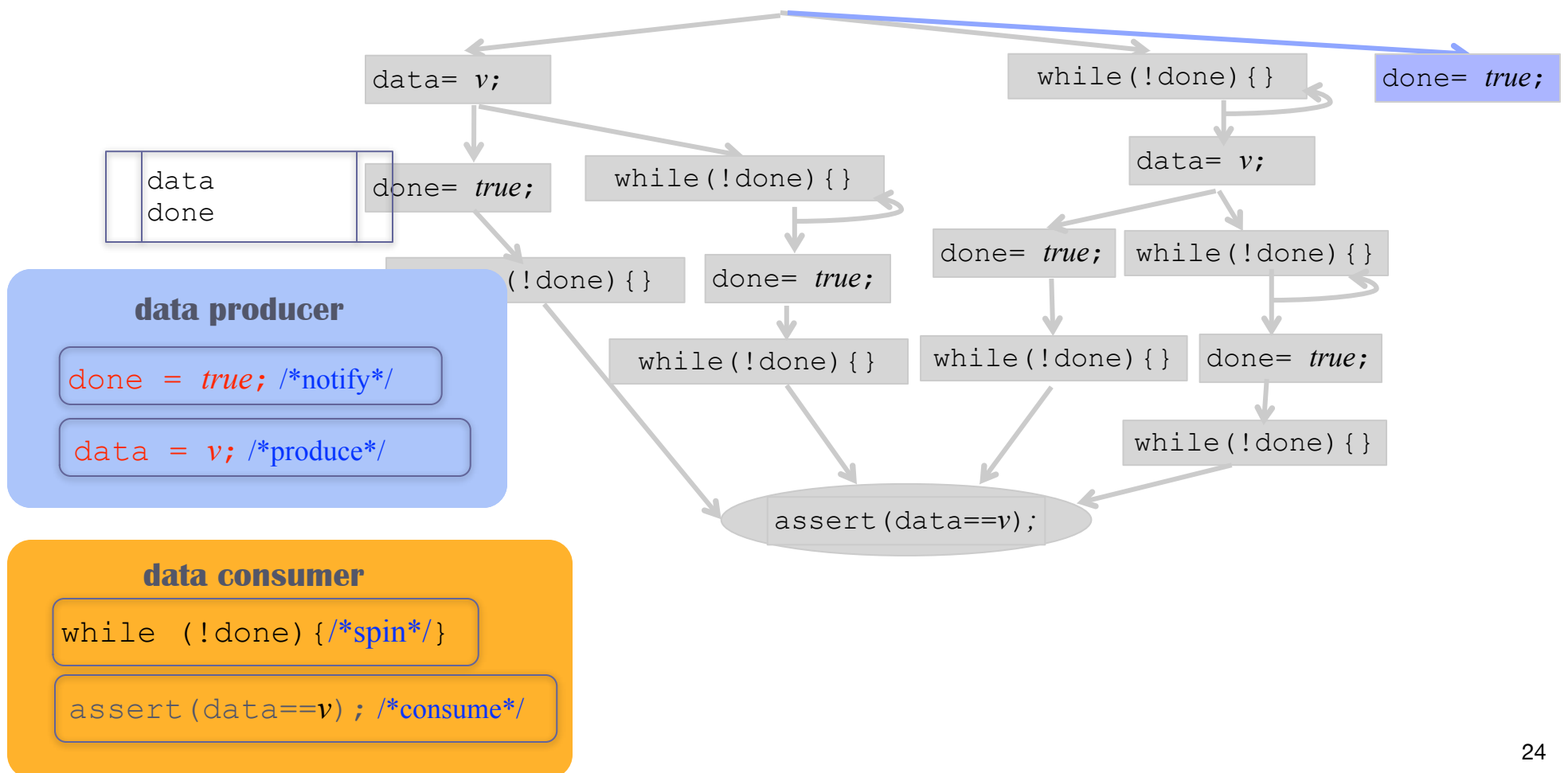
# Motivating Example



# Motivating Example

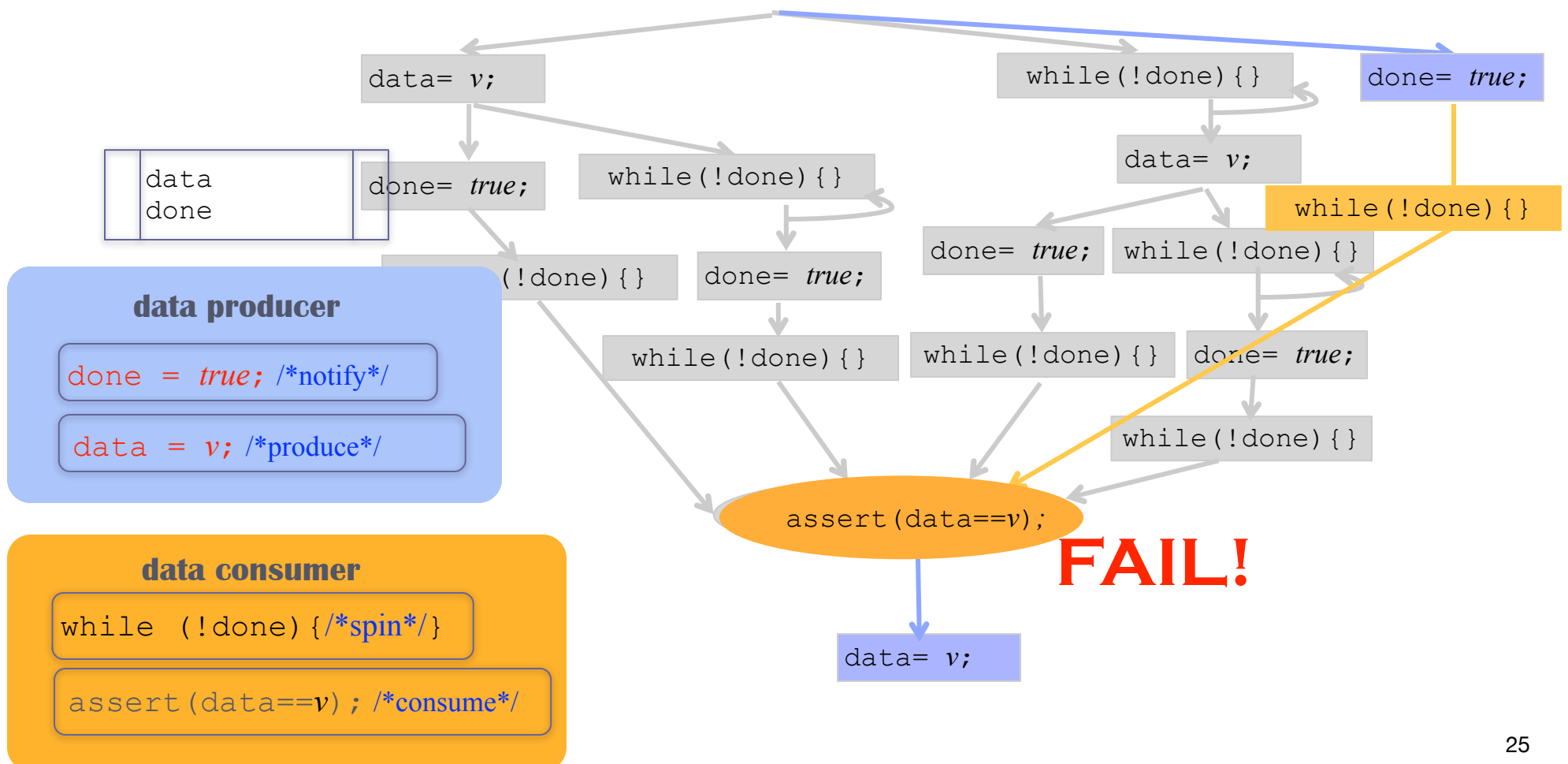


# Motivating Example





# 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

```
class SimpleSync
```

	<pre>int data=0; boolean done=false;</pre>	
--	--	--

## data producer

```
data = v; /*produce*/  
synchronized (this)  
{ done = true; /*notify*/ }
```

## data consumer

```
boolean r = false;  
while (!r) /*spin*/  
  synchronized (this)  
  { r = done; }  
assert (data==v) ; /*consume*/
```

# Motivating Example

---

(1) use a synchronization

```
class SimpleSync
```

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

## data producer

```
data = v; /*produce*/  
synchronized(this)  
{ done = true; /*notify*/ }
```

lock

unlock

## data consumer

```
boolean r = false;  
while (!r) /*spin*/  
synchronized(this)  
{ r = done; }  
assert(data==v) /*consume*/
```

lock

unlock

# Motivating Example

---

(1) use a synchronization

```
class SimpleSync
```

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

**data producer**

```
data = v; /*produce*/  
synchronized(this)  
{ done = true; /*notify*/ }
```

lock

unlock

**data consumer**

```
boolean r = false;  
while (!r) /*spin*/  
synchronized(this)  
{ r = done; }  
assert(data==v) ; /*consume*/
```

lock

unlock

# Motivating Example

---

(1) use a synchronization

```
class SimpleSync
```

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

**data producer**

```
data = v; /*produce*/  
synchronized(this)  
{ done = true; /*notify*/ }
```

lock

unlock

**data consumer**

```
boolean r = false;  
while (!r) /*spin*/  
synchronized(this)  
{ r = done; }  
assert(data==v) ; /*consume*/
```

lock

unlock

# Motivating Example

---

(2) use a memory barrier using “volatile” keyword

```
class SimpleVolatile
```

	<pre>int data=0; boolean done=false;</pre>	
--	--	--

## 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
```

<pre>int data=0; volatile boolean done=false;</pre>
---

## 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

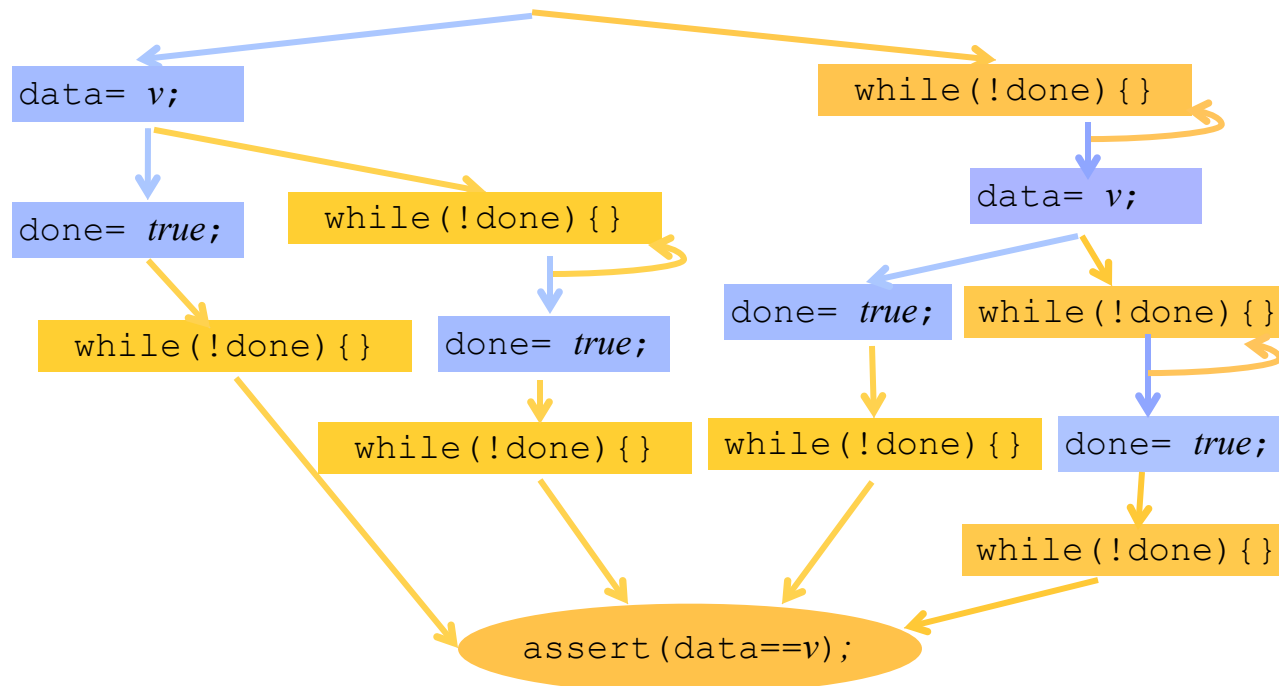
- all of its atomic actions occur in some global order  
(consistent with the program order on each thread)
- all threads see values written to main memory in a consistent order

## Relaxed Memory Model

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

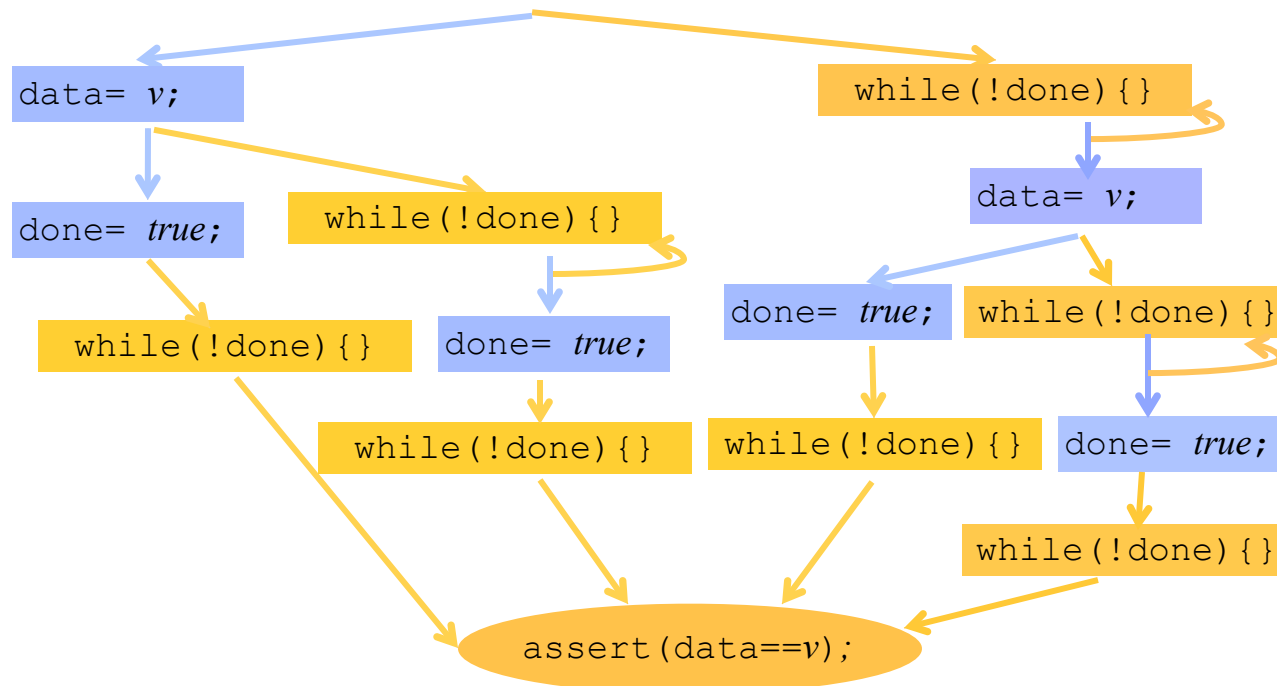
# Java Memory Model

- 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

---

- 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)

```
class Simple
```

	<pre>int data=0; boolean done=false;</pre>	
--	--	--

**main**

```
start data producer, start data consumer
```

```
join data producer, join data consumer
```

**data producer**

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

**data consumer**

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

# Java Memory Model

→ **program order** (intra-thread)

```
class Simple
```

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

**main**

```
start data producer, → start data consumer  
join data producer, → join data consumer
```

**data producer**

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

**data consumer**

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

# Java Memory Model

---

➔ **program order** (intra-thread)

➔ **synchronize-with order** (inter-thread)

```
class Simple
```

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

**main**

```
start data producer, start data consumer
```

```
join data producer, join data consumer
```

**data producer**

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

**data consumer**

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

# Java Memory Model

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

```
class Simple
```

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

**main**

```
start data producer, start data consumer  
start of thread  
join data producer, join data consumer
```

**data producer**

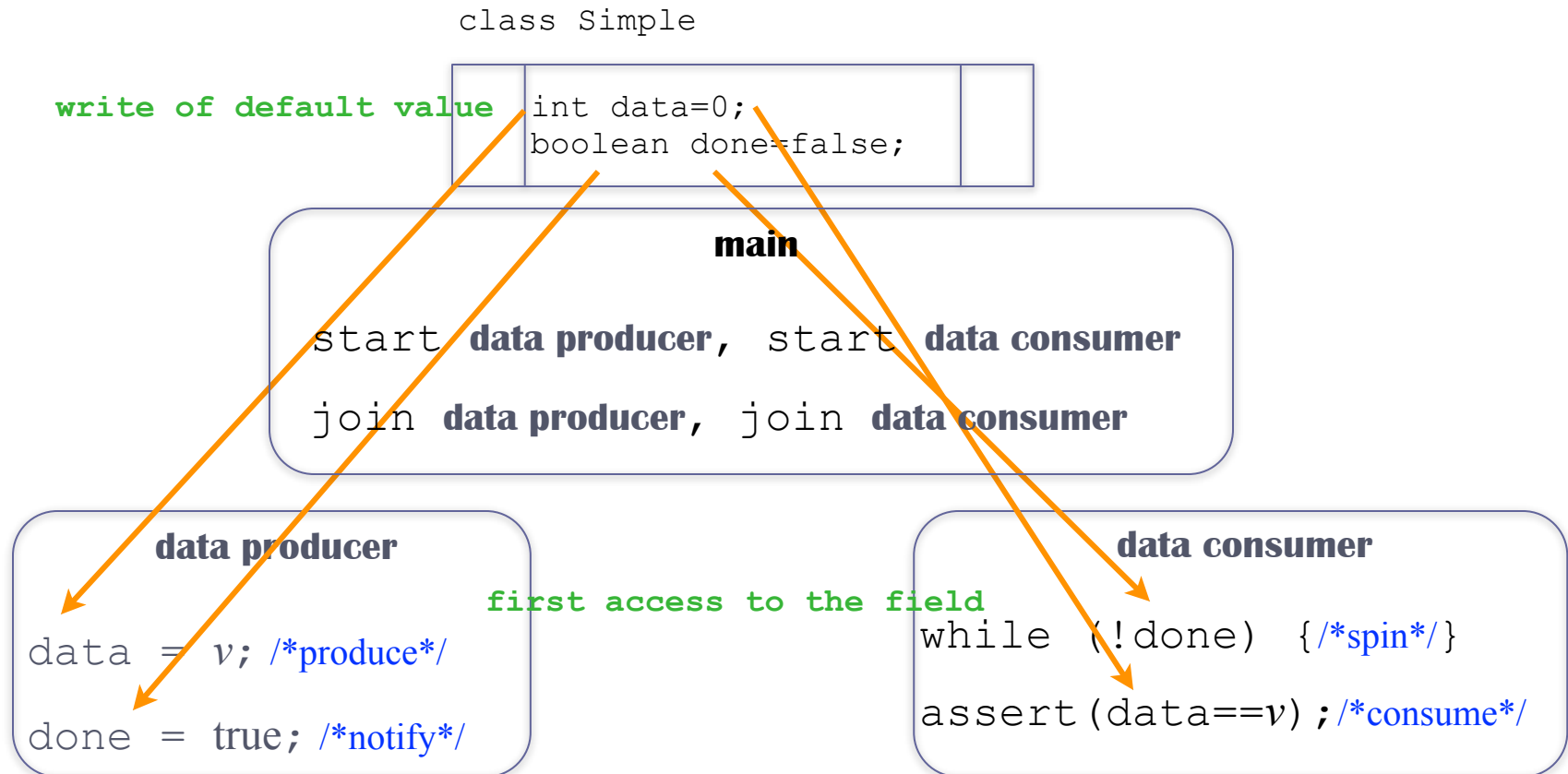
```
first action of thread  
data = v; /*produce*/  
done = true; /*notify*/
```

**data consumer**

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

# Java Memory Model

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



# Java Memory Model

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

```
class Simple
```

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

**main**

```
start data producer, start data consumer  
join data producer, join data consumer
```

**data producer**

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

write of volatile

read of volatile

**data consumer**

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

# Java Memory Model

→ **program order** (intra-thread)

→ **synchronize-with order** (inter-thread)

```
class Simple
```

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

**main**

```
start data producer, start data consumer  
join data producer, join data consumer
```

**data producer**

```
data = v; /*produce*/  
synchronized(this)  
{ done = true; /*notify*/ }  
monitor enter  
monitor exit
```

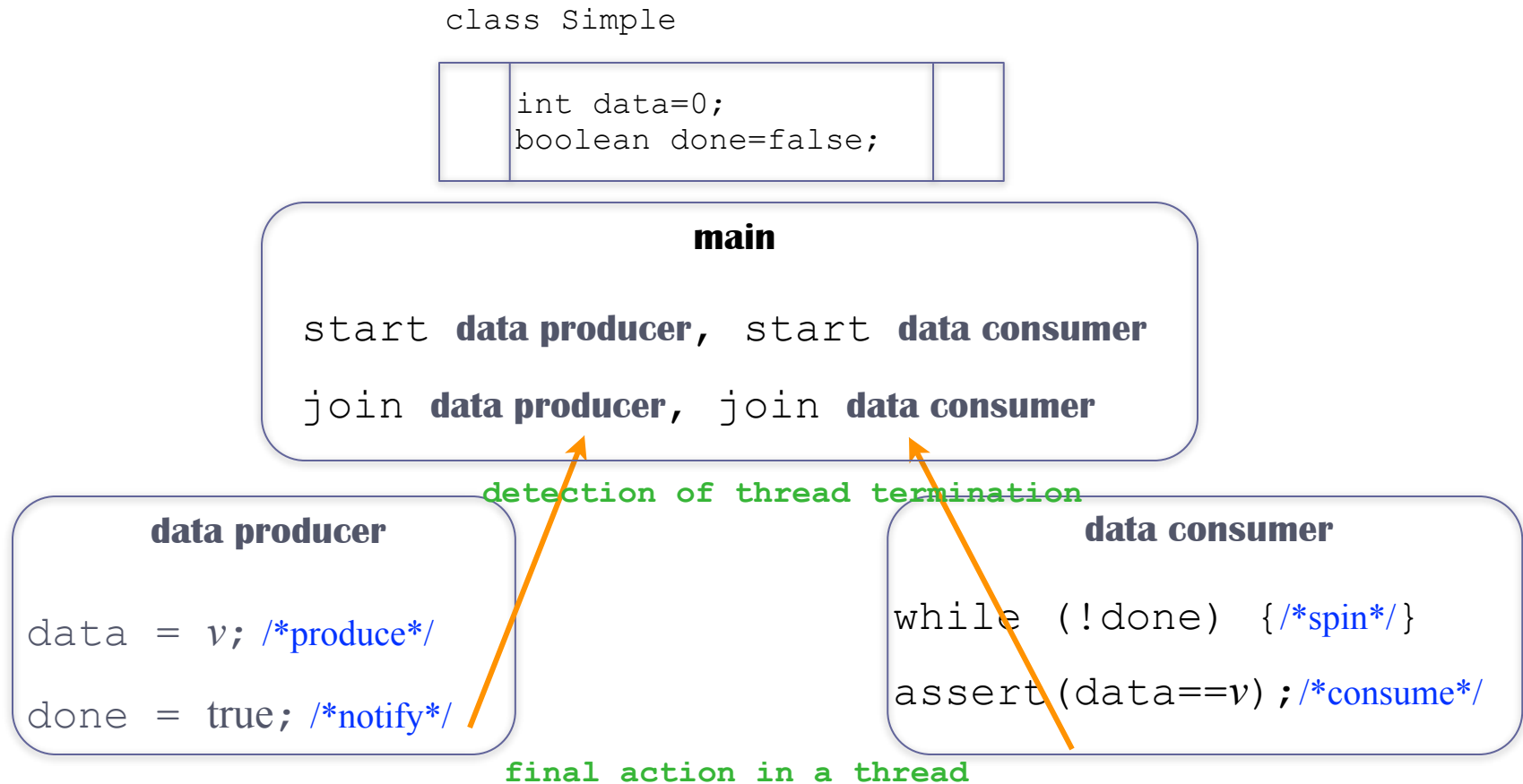
**data consumer**

```
boolean r = false;  
while (!r) /*spin*/  
synchronized(this)  
{ r = done; }  
assert (data==v) ; /*consume*/  
monitor enter  
monitor exit
```



# Java Memory Model

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



# Java Memory Model

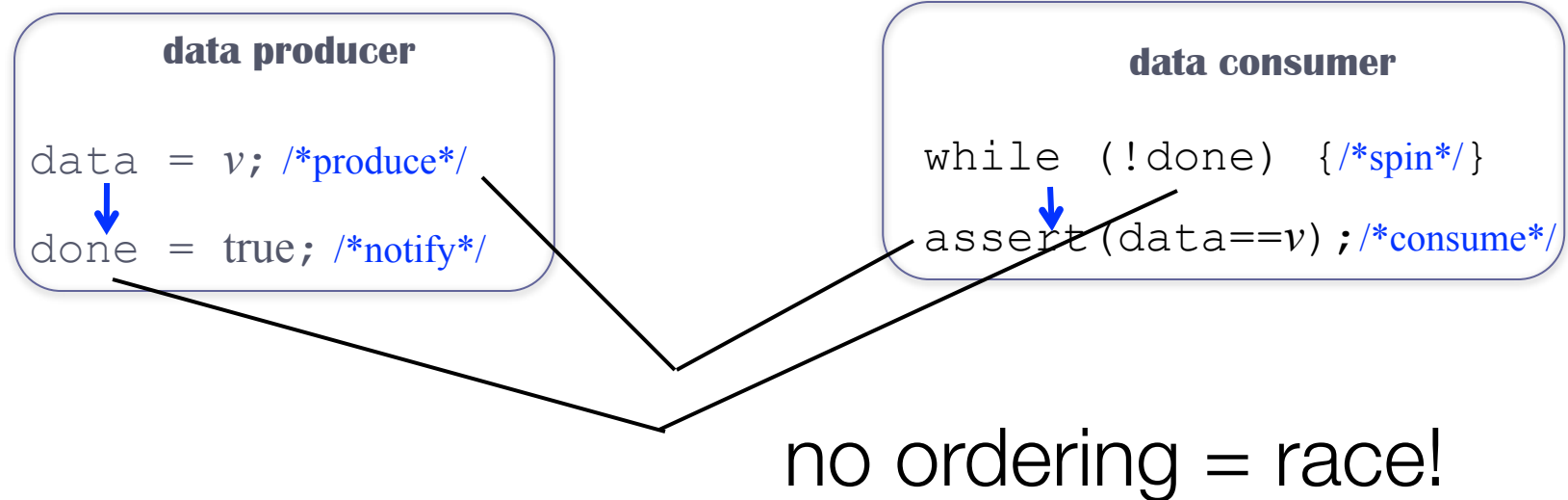
---

- 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

---

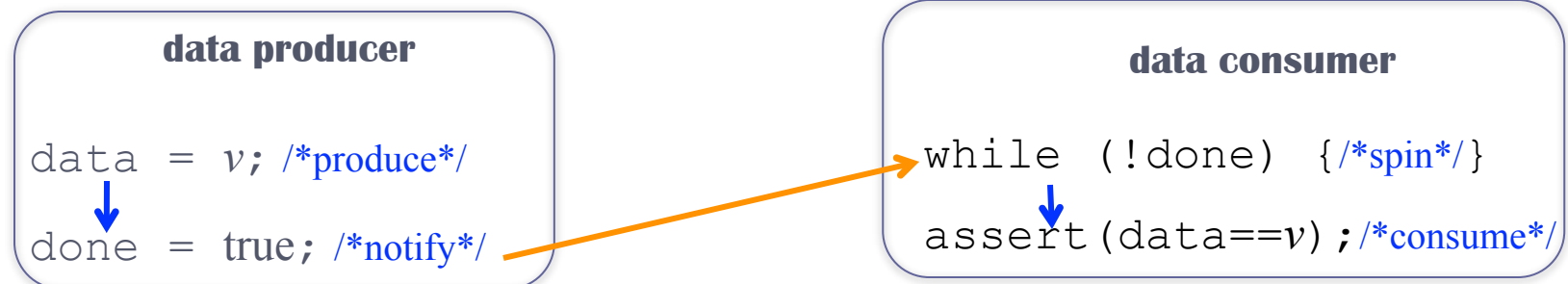
- Data Races in `Simple` class



# 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  $\rightarrow$  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

read  $r$  of variable  $v$   
is allowed to see  
the result of a write  $w$

$$w = W(r)$$

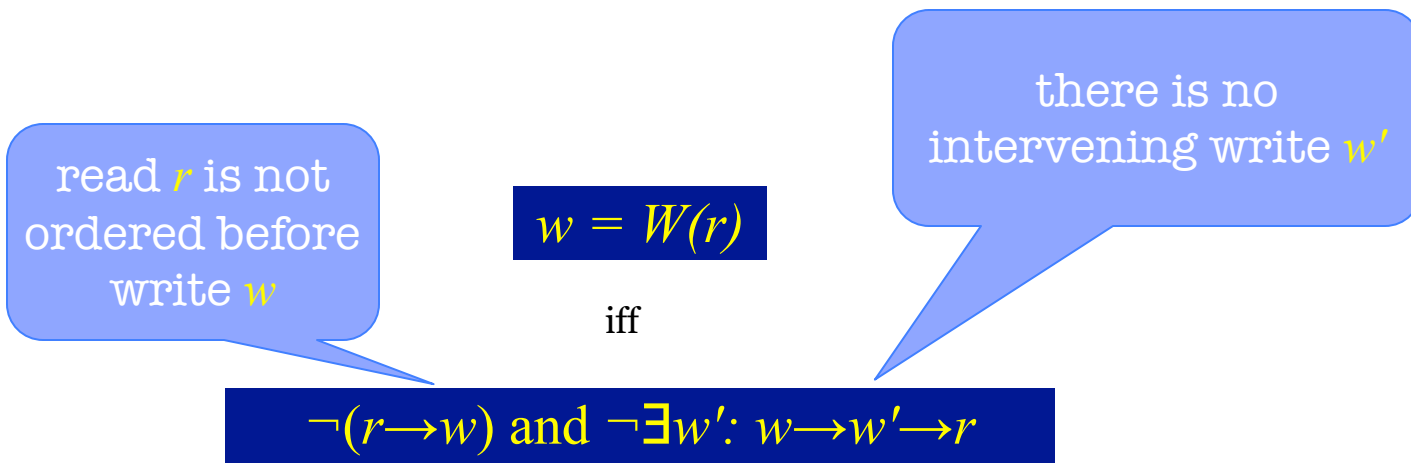
iff

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

# Java Memory Model

---

- Happens-before consistency
  - : determines the values a non-volatile read can see





# Java Memory Model

---

- 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

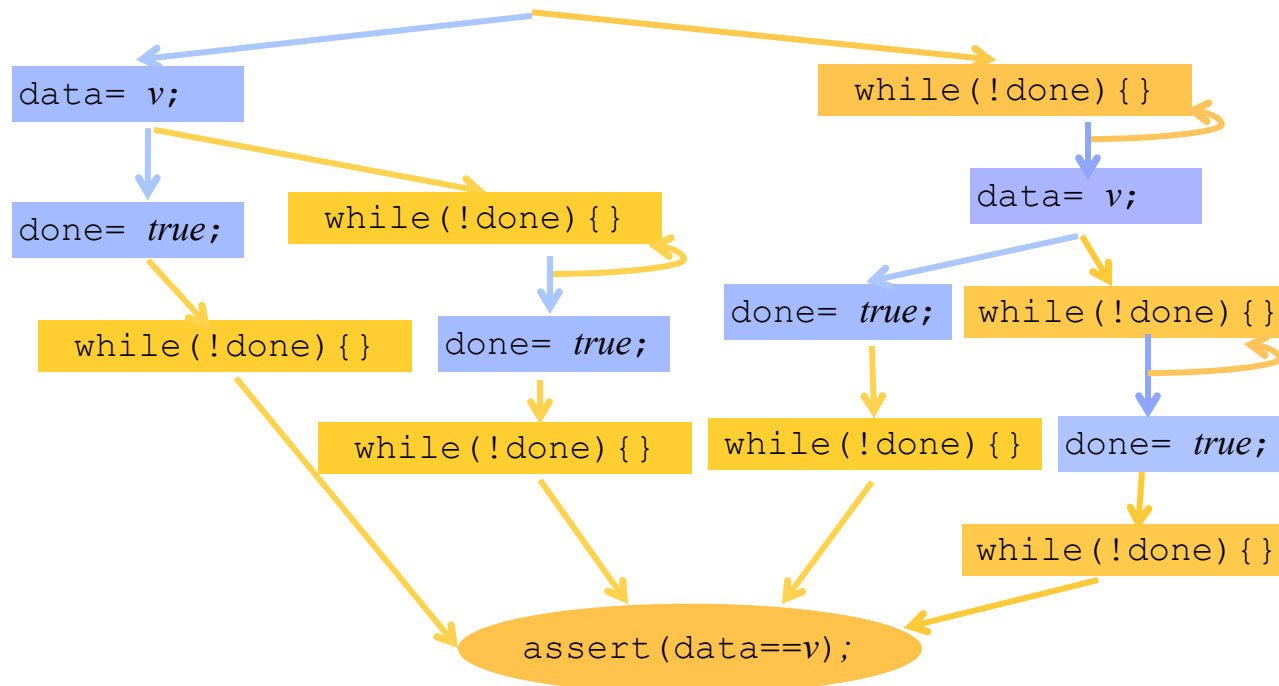
**Programs whose SC executions have no races  
must have only SC executions.**

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**

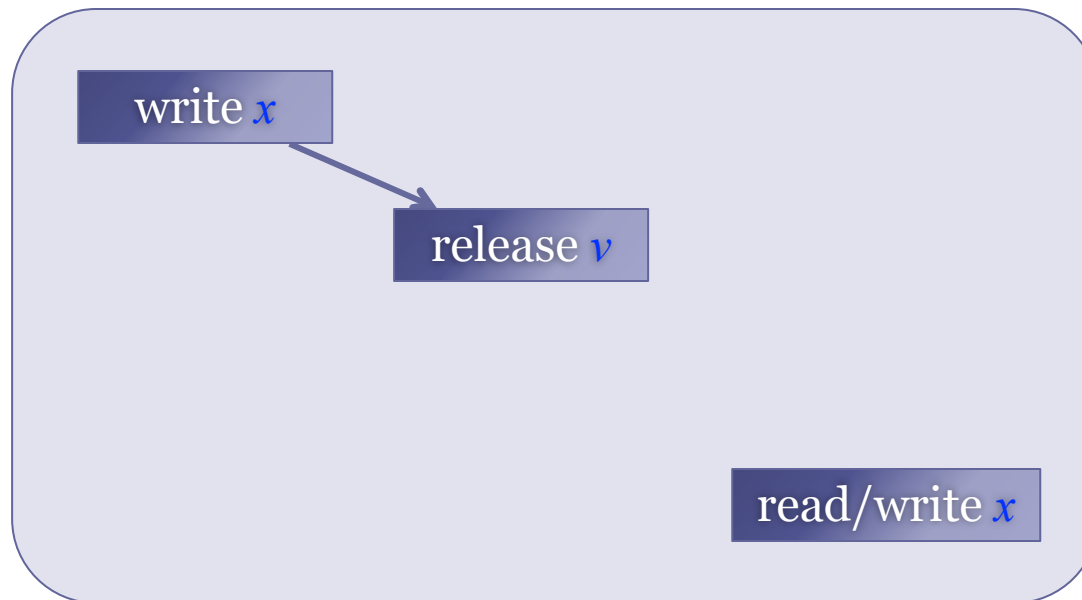
a **set of memory locations**  
whose recent update  
happens-before the  
release of  **$v$**

# Algorithm

---

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

$$h(v) = X$$

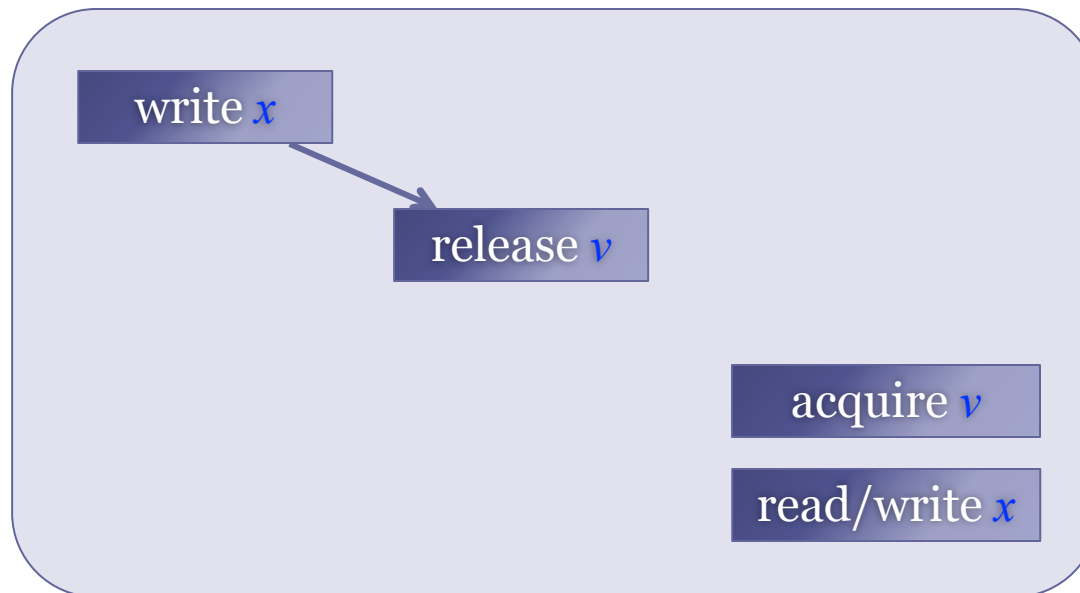


# Algorithm

---

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

$$h(v) = X$$

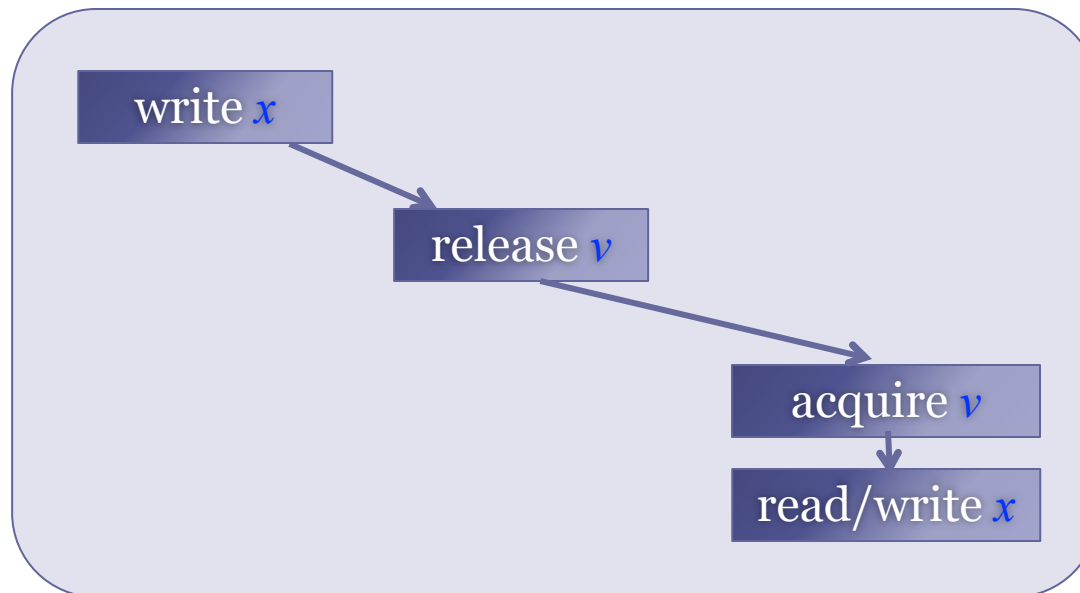


# Data Race Detection Algorithm

---

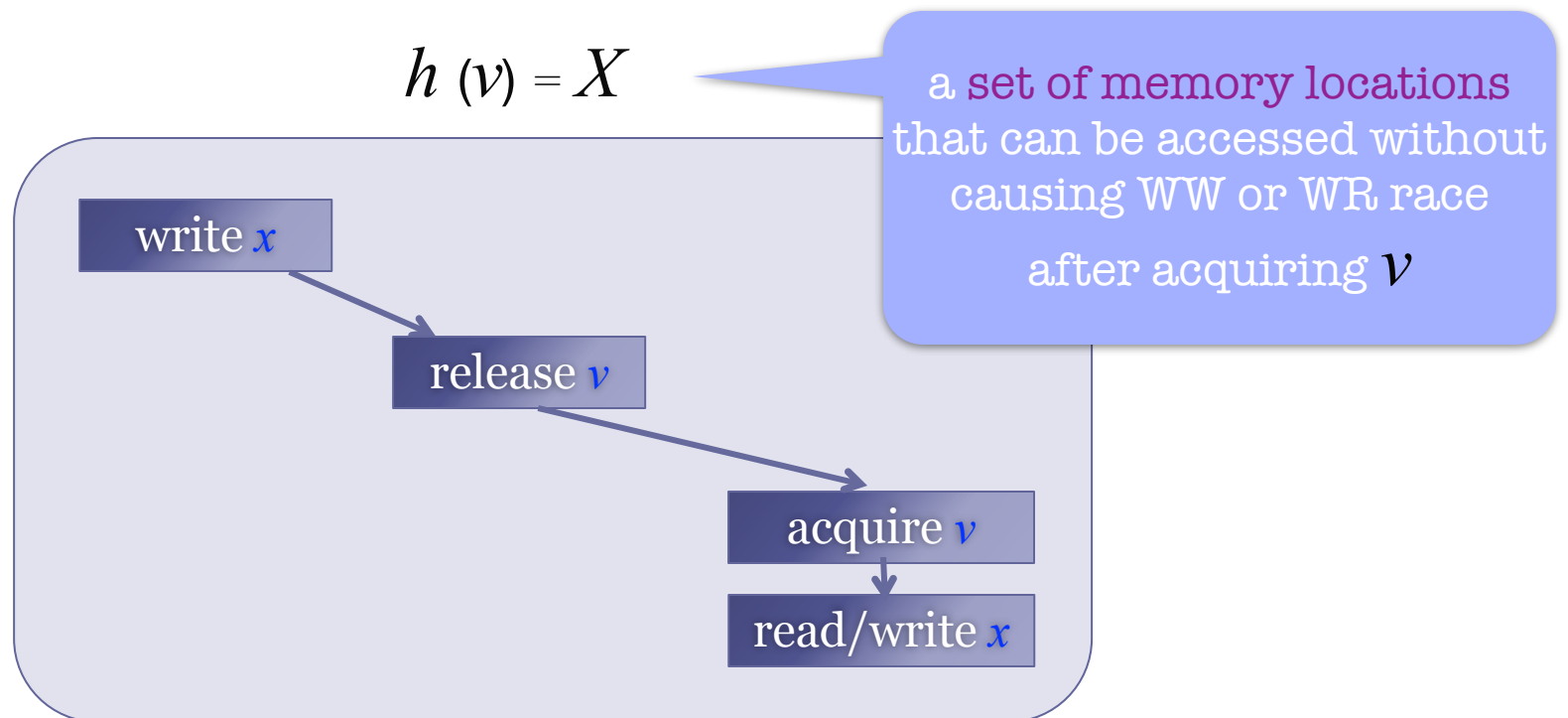
- Summary Function  $h$ 
  - Happens-before relation is transitive and formed by *release-acquire* pair
  - summarize the happens-before relation as a function

$$h(v) = X$$



# Data Race Detection 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

# Data Race Detection Algorithm

---

- Summary Function  $h$

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

acquire

release

invalidate

norace

# Data Race Detection Algorithm

---

- Summary Function  $h$

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

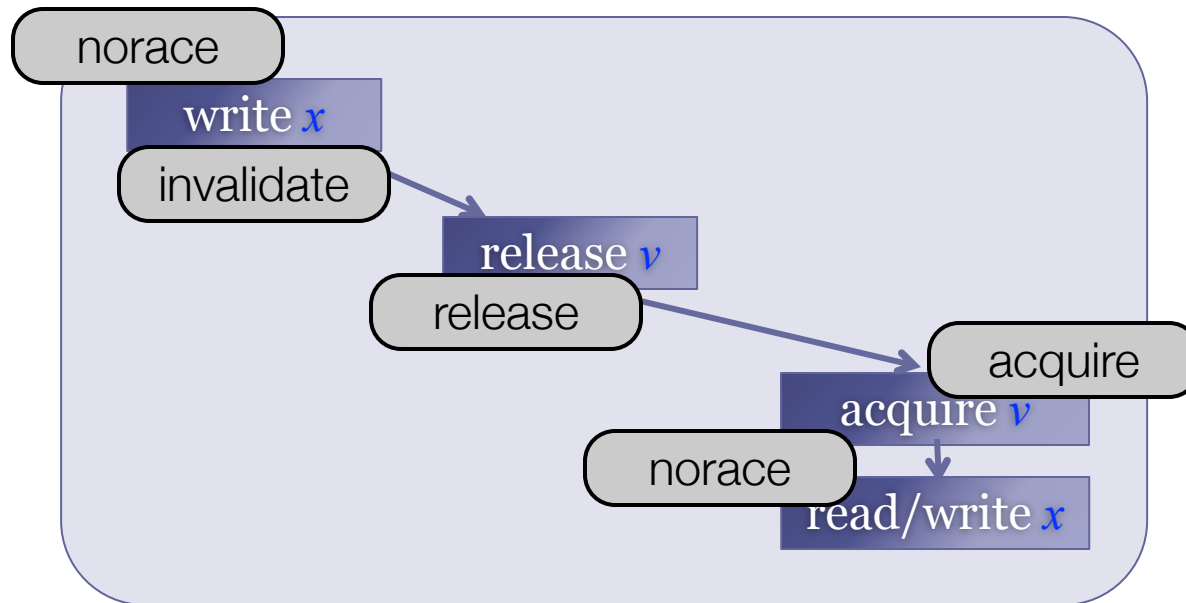
acquire	: volatile read, lock, ...
release	: volatile write, unlock, ...
invalidate	: non-volatile write
norace	: non-volatile read/write



# Data Race Detection Algorithm

- Summary Function  $h$

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



# 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

**main**

```
class Simple
```

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

```
start data producer, data consumer;
```

```
data= v;
```

```
done= true;
```

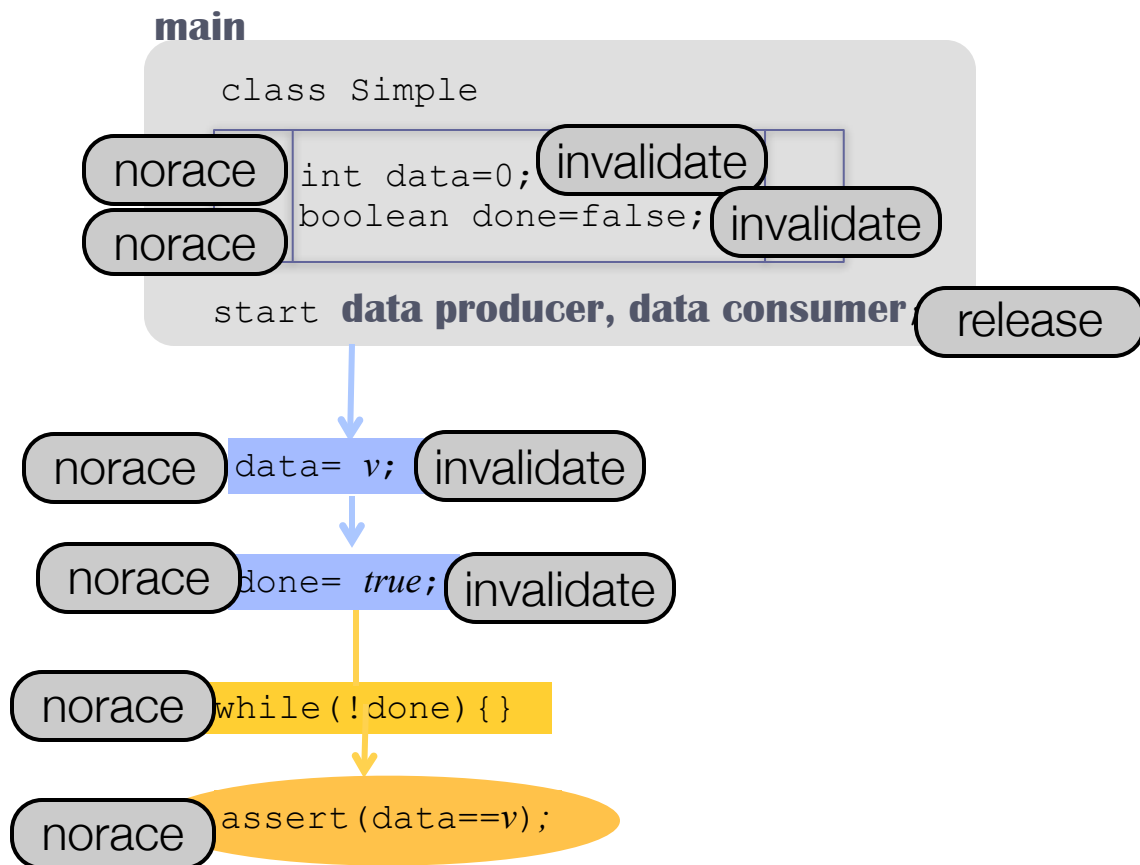
```
while (!done) {}
```

```
assert (data==v);
```

data race

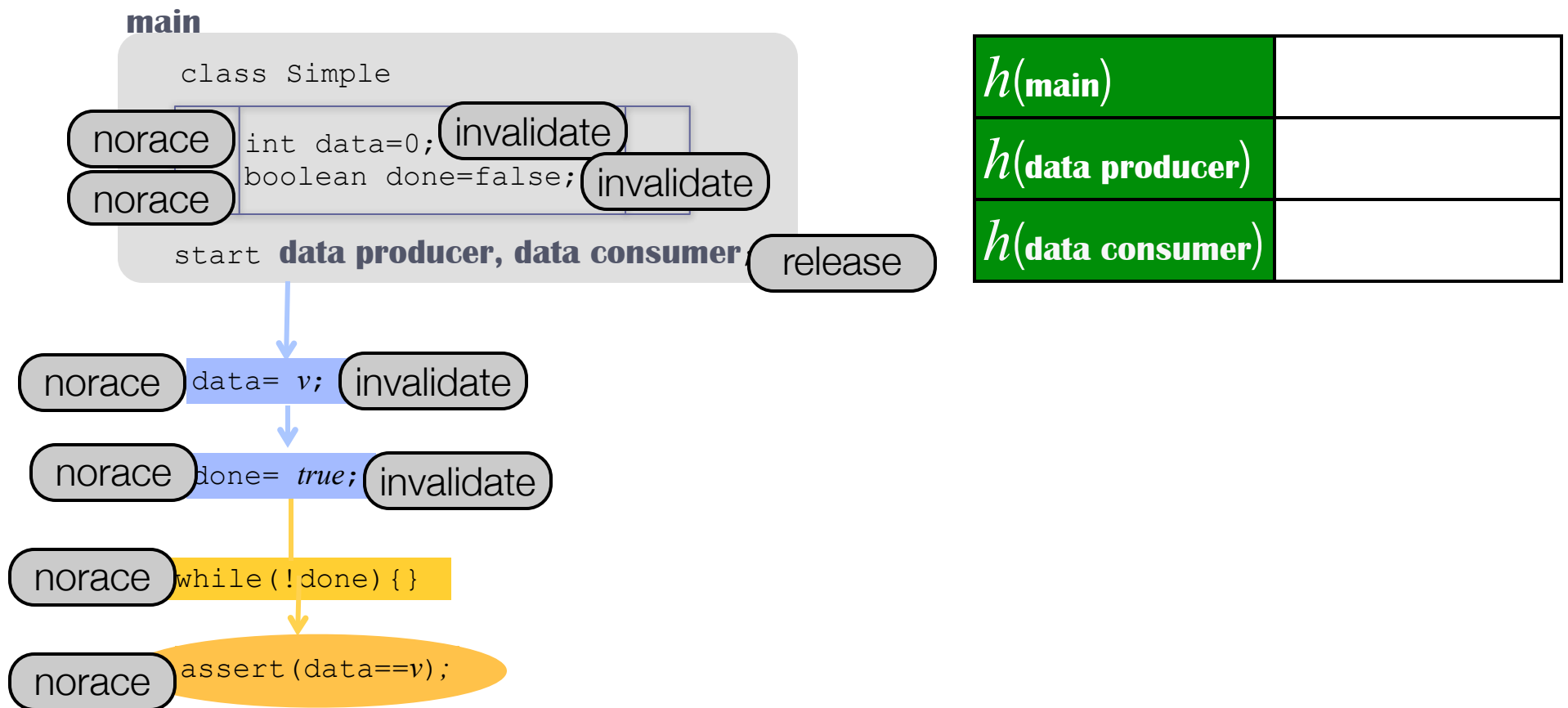
# Data Race Detection Algorithm

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



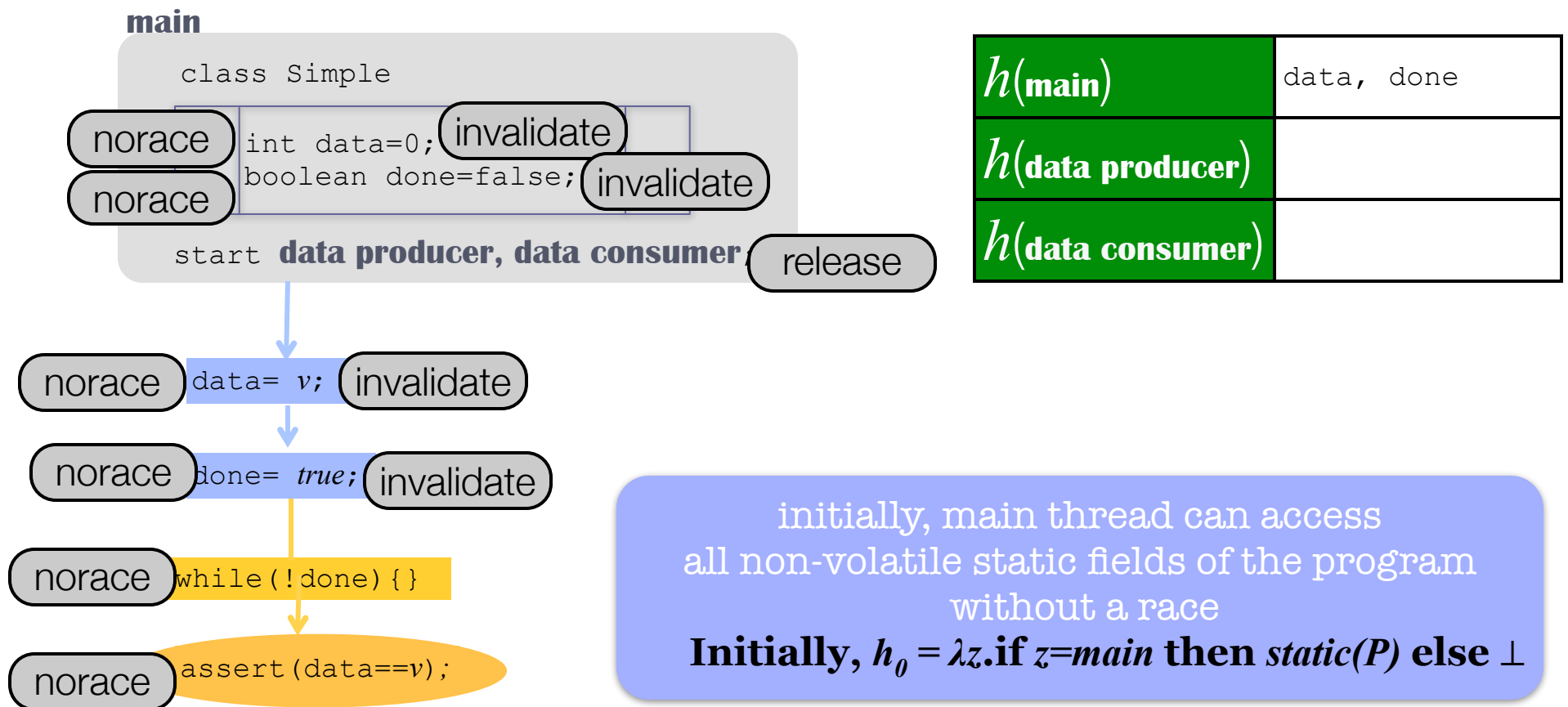
# Data Race Detection Algorithm

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



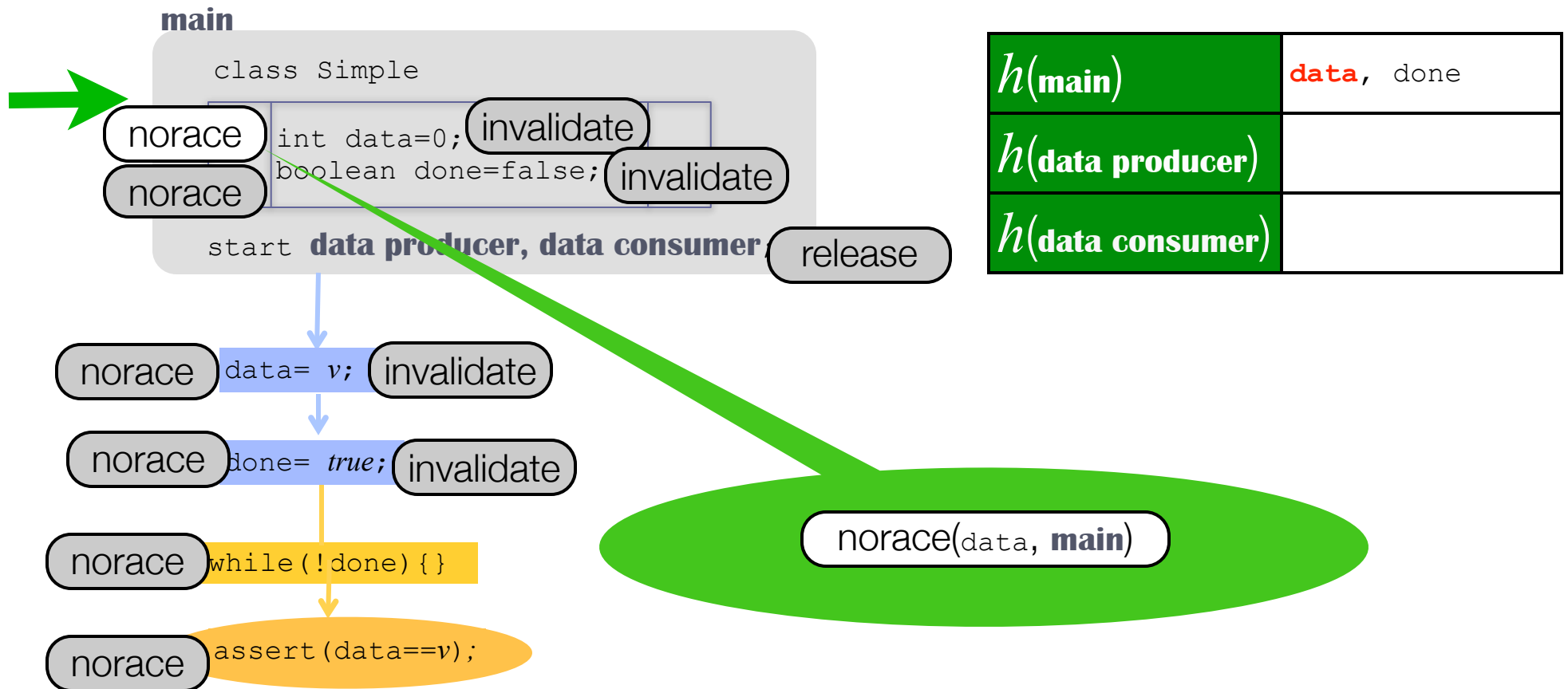
# Data Race Detection Algorithm

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



# Data Race Detection Algorithm

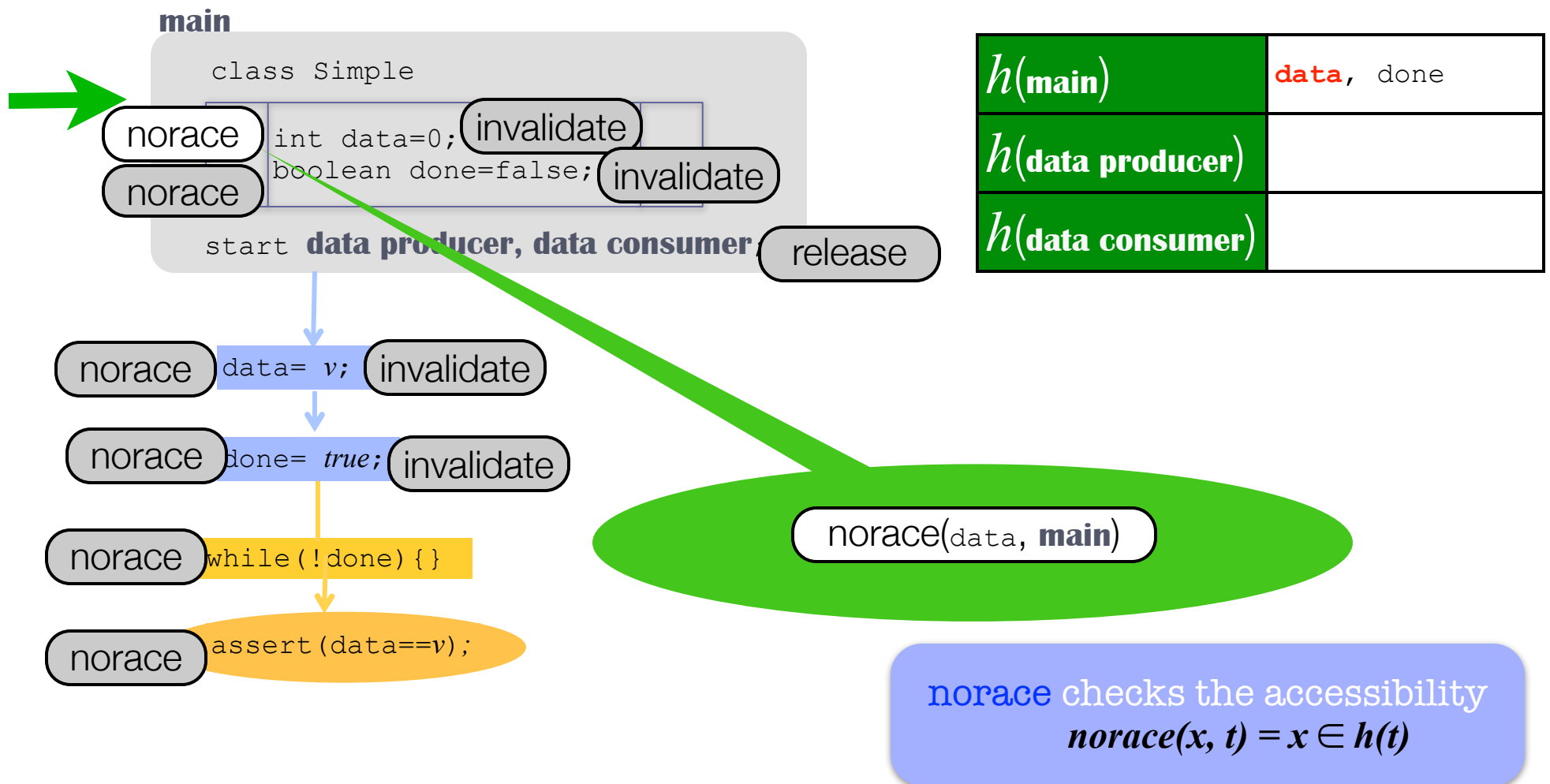
- Data race detection using summary function  $h$  for Simple java program





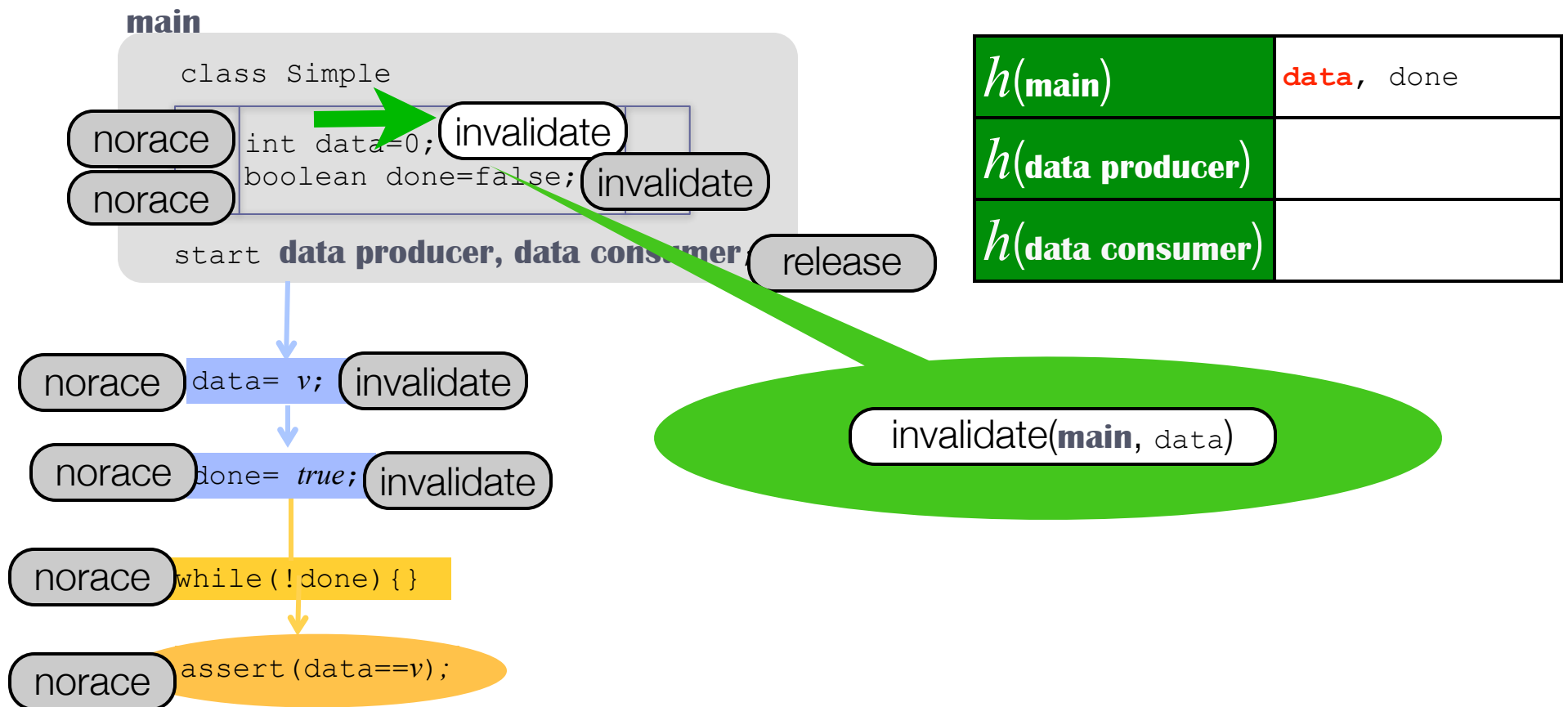
# Data Race Detection Algorithm

- Data race detection using summary function  $h$  for simple java program



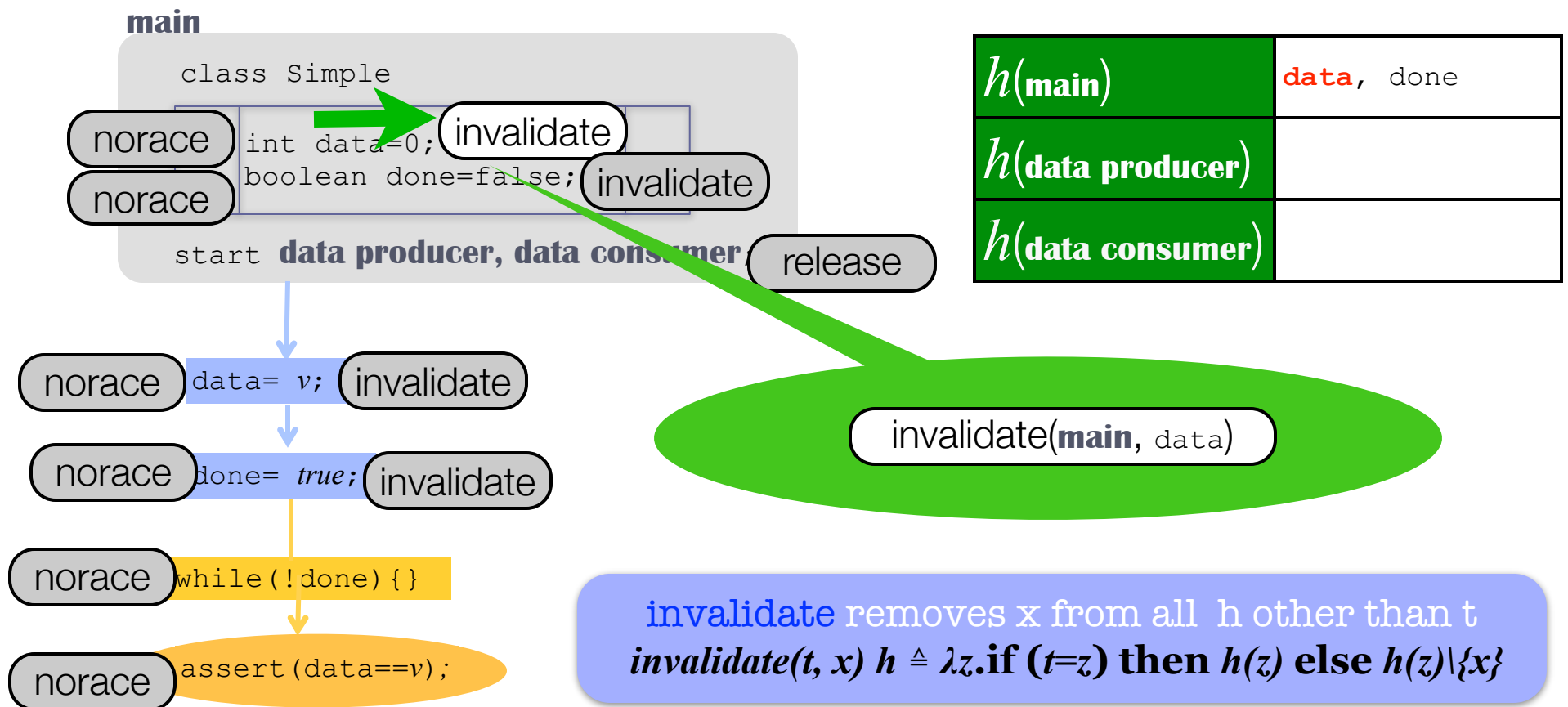
# Data Race Detection Algorithm

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



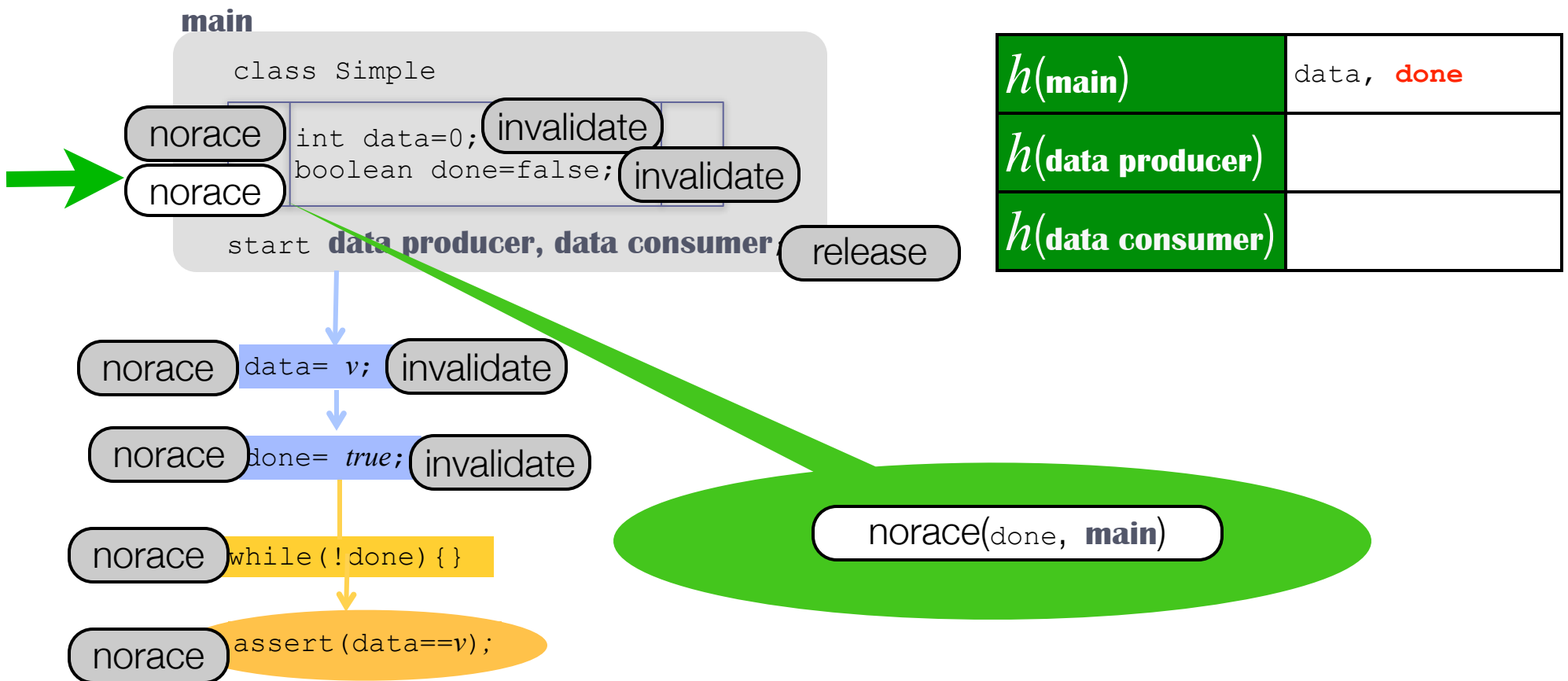
# Data Race Detection Algorithm

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



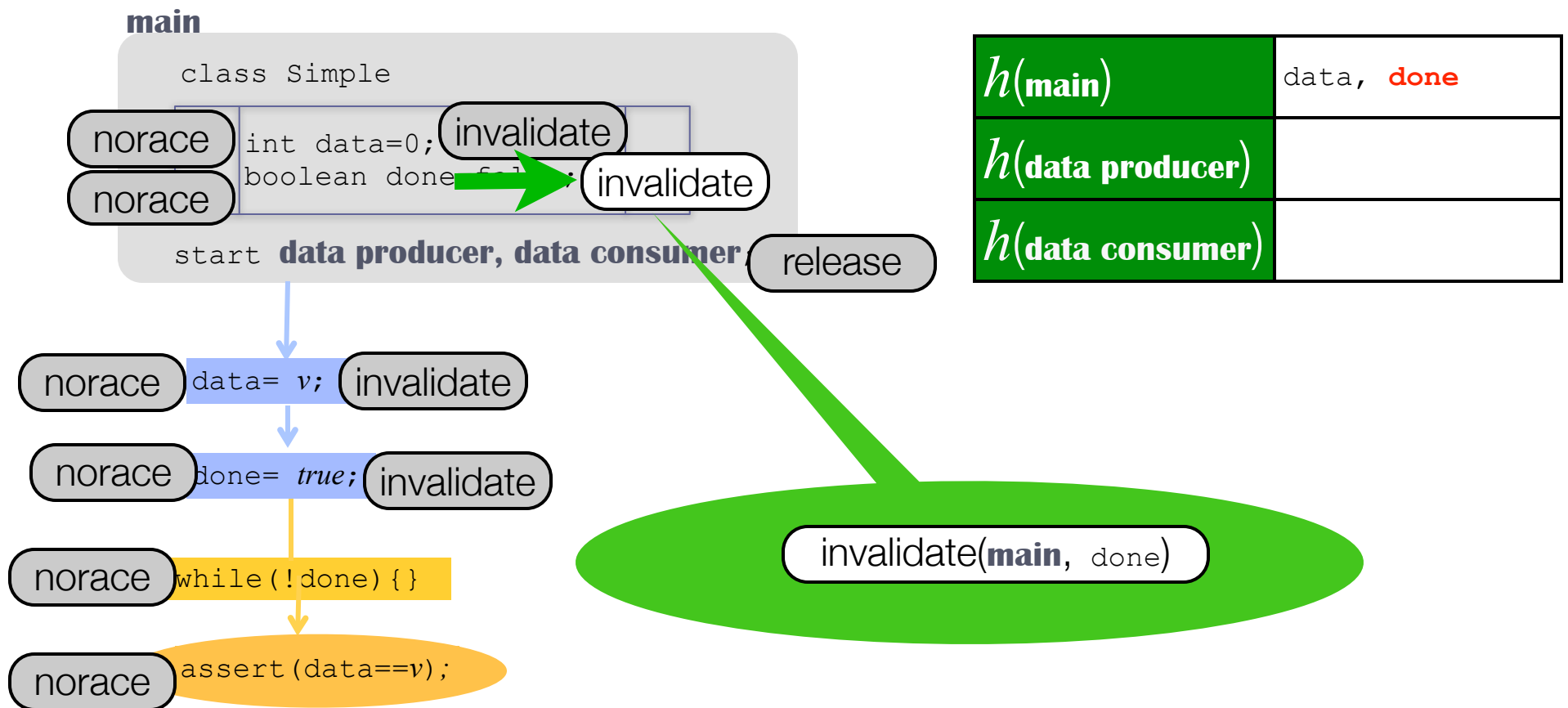
# Data Race Detection Algorithm

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



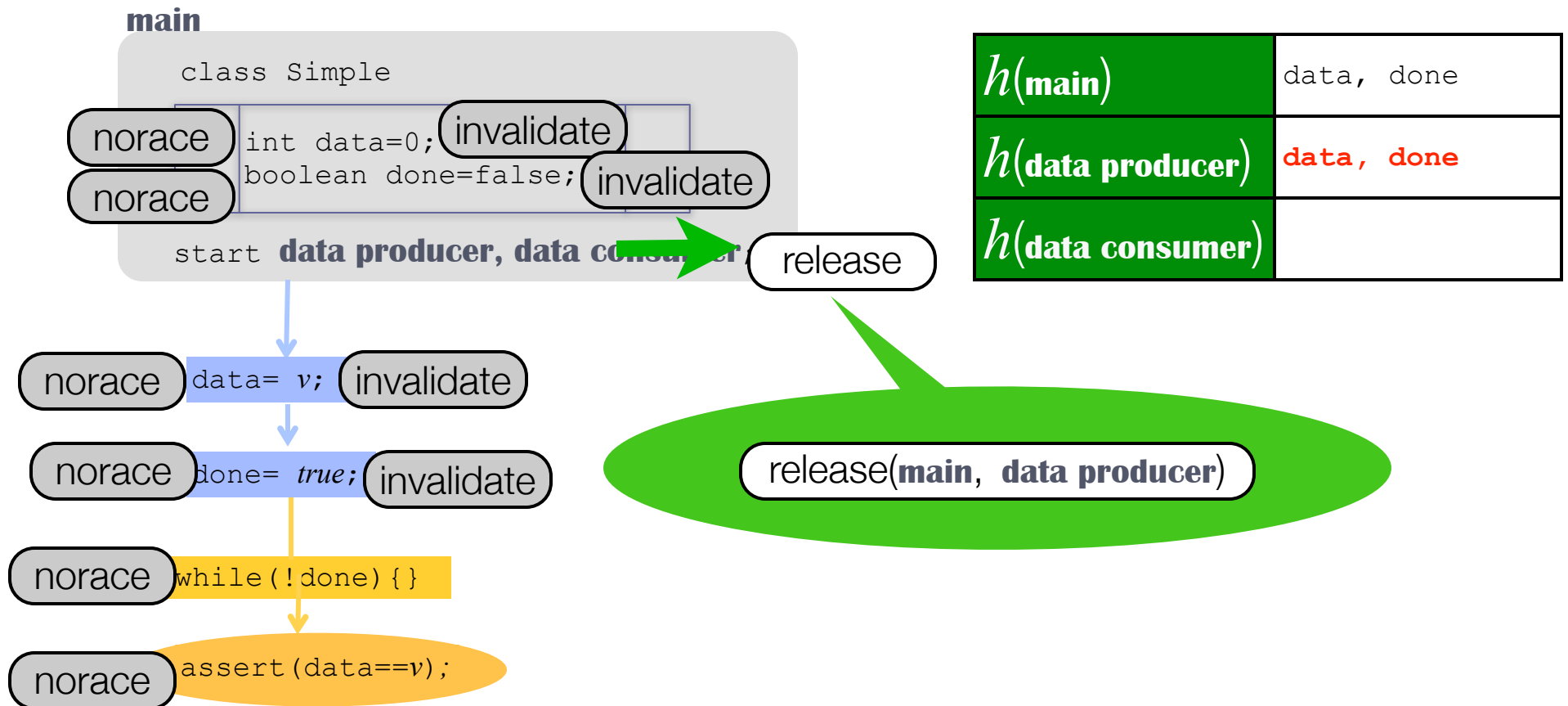
# Data Race Detection Algorithm

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



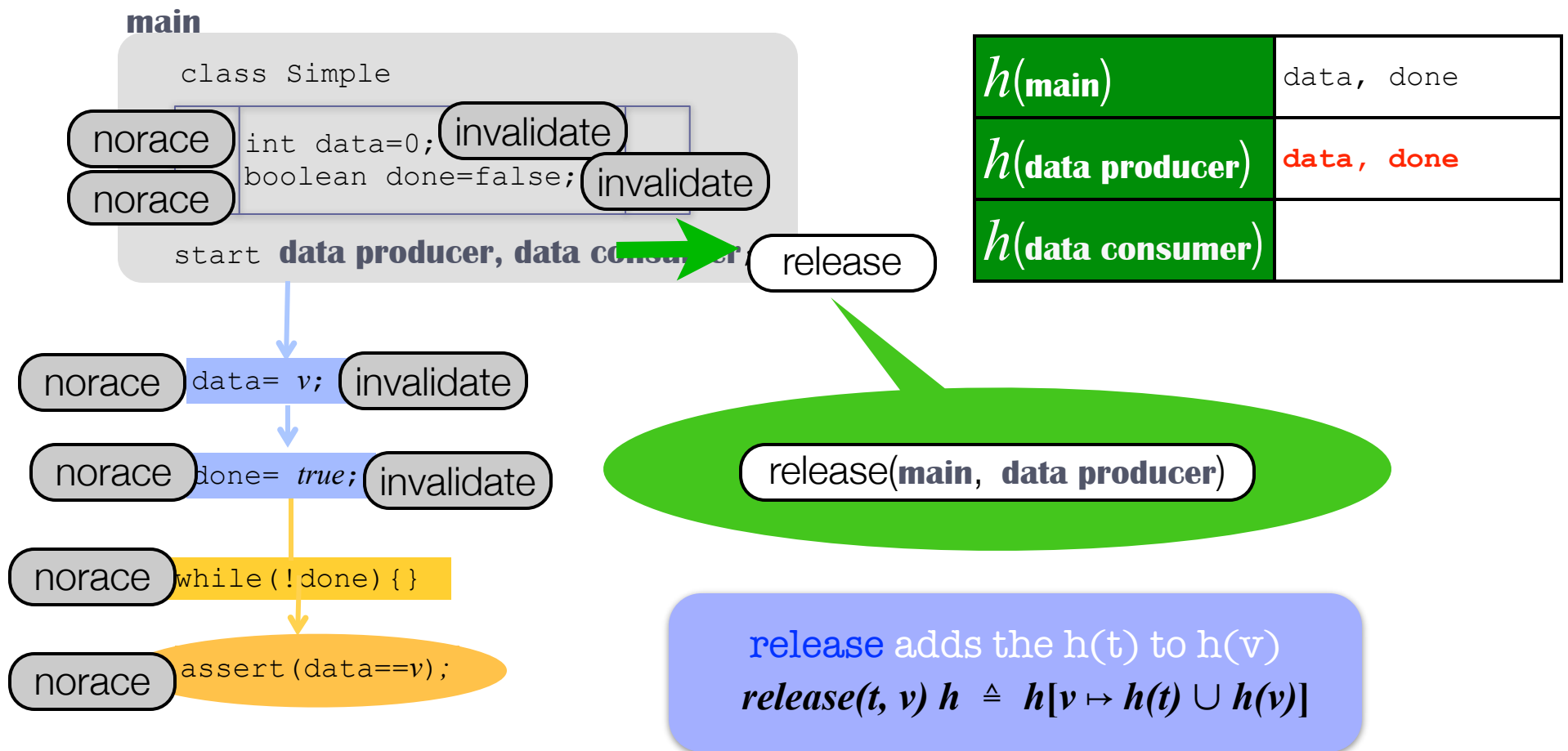
# Data Race Detection Algorithm

- Data race detection using summary function  $h$  for simple java program



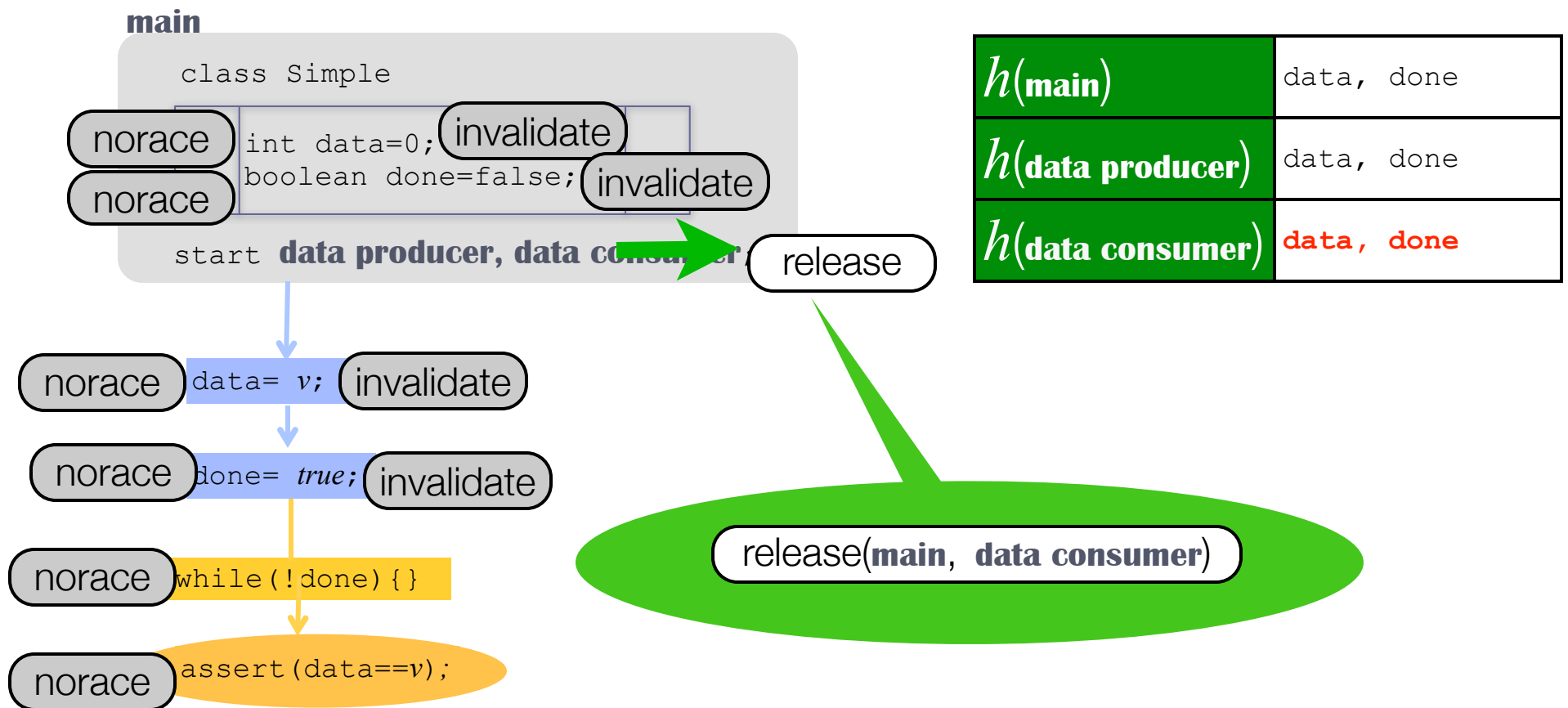
# Data Race Detection Algorithm

- 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





# Data Race Detection Algorithm

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

**main**

```
class Simple
```

norace

```
int data=0; invalidate
```

norace

```
boolean done=false; invalidate
```

```
start data producer, data consumer release
```

$h(\text{main})$

data, done

$h(\text{data producer})$

**data**, done

$h(\text{data consumer})$

data, done

norace

```
data= v; invalidate
```

norace

```
done= true; invalidate
```

norace

```
while (!done) {}
```

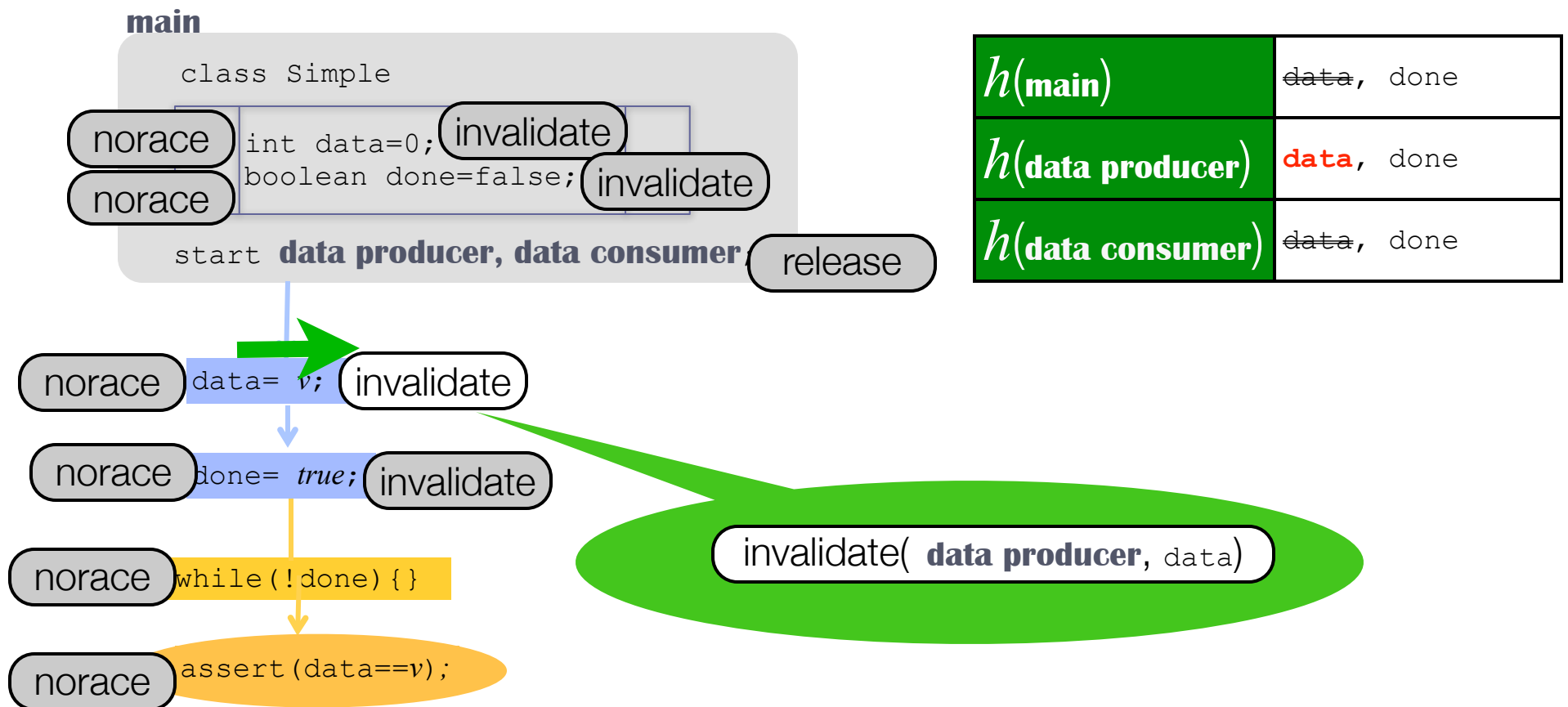
norace

```
assert (data==v);
```

norace(data, **data producer**)

# 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

**main**

```
class Simple
```

norace

```
int data=0; invalidate
```

norace

```
boolean done=false; invalidate
```

```
start data producer, data consumer release
```

$h(\mathbf{main})$	done
$h(\mathbf{data\ producer})$	data, <b>done</b>
$h(\mathbf{data\ consumer})$	done

norace

```
data= v; invalidate
```

norace

```
done= true; invalidate
```

norace

```
while (!done) {}
```

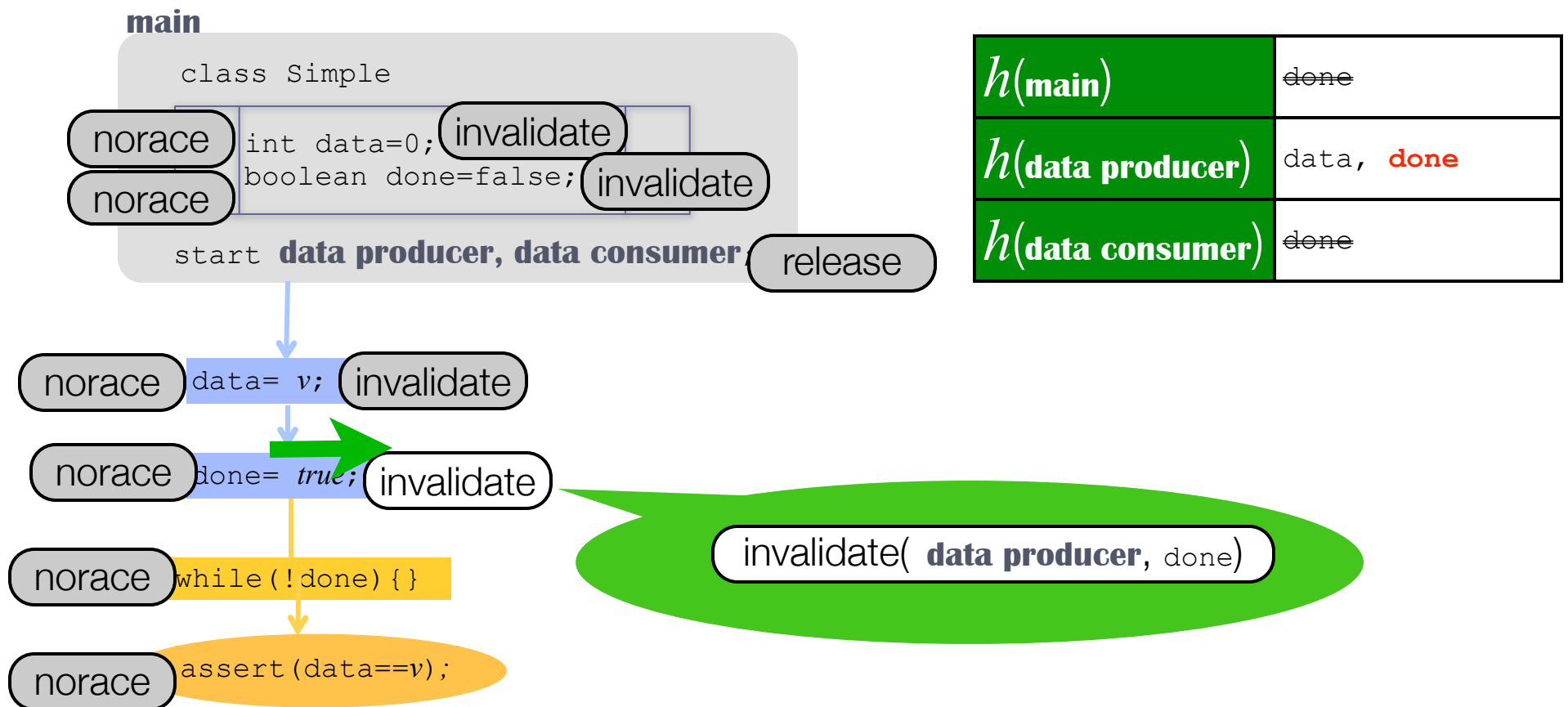
norace

```
assert (data==v);
```

norace(done, **data producer**)

# 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

**main**

```
class Simple
```

norace

```
int data=0; invalidate
```

norace

```
boolean done=false; invalidate
```

```
start data producer, data consumer release
```

$h(\text{main})$	
$h(\text{data producer})$	data, done
$h(\text{data consumer})$	<b>FAIL!</b>

norace

```
data= v; invalidate
```

norace

```
done= true; invalidate
```

norace

```
while (!done) {
```

norace(done, data consumer)

norace

```
assert (data==v);
```

# Data Race Detection Algorithm

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

**main**

```
class Simple
```

norace

```
int data=0; invalidate
```

norace

```
boolean done=false; invalidate
```

```
start data producer, data consumer release
```

$h(\text{main})$	
$h(\text{data producer})$	data, done
$h(\text{data consumer})$	<b>FAIL!</b>

norace

```
data= v; invalidate
```

norace

```
done= true; invalidate
```

norace

```
while (!done) {}
```

norace

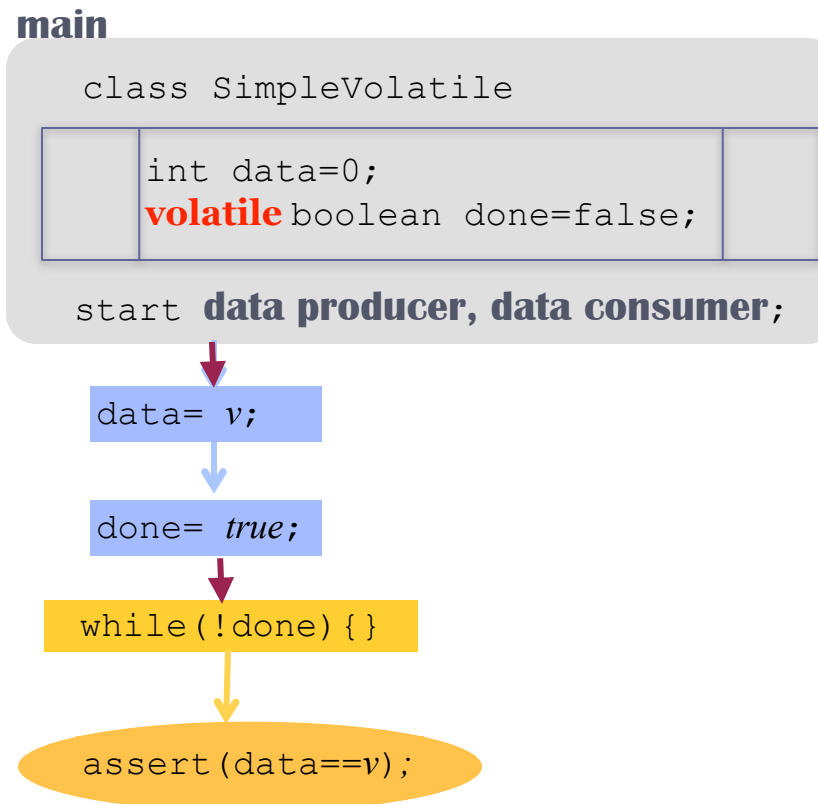
```
assert (data==v);
```

norace(data, data consumer)

# Data Race Detection Algorithm

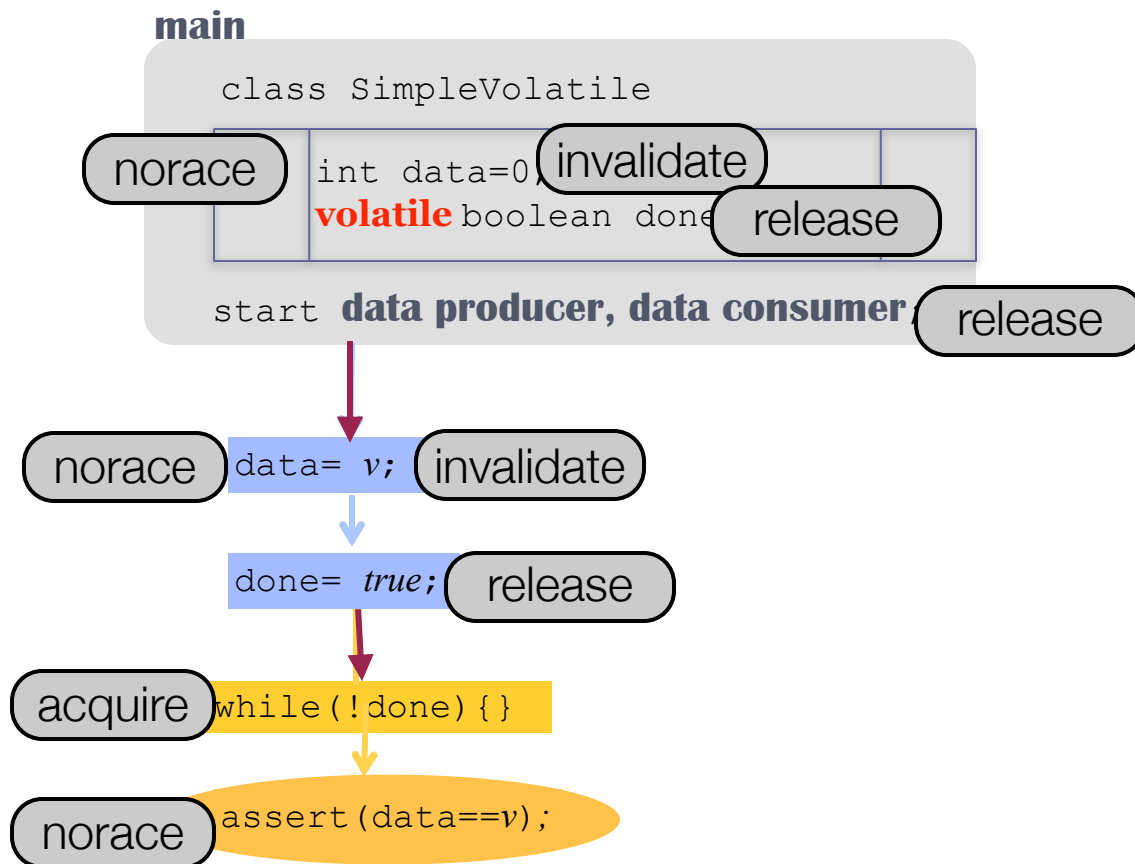
---

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



# Data Race Detection Algorithm

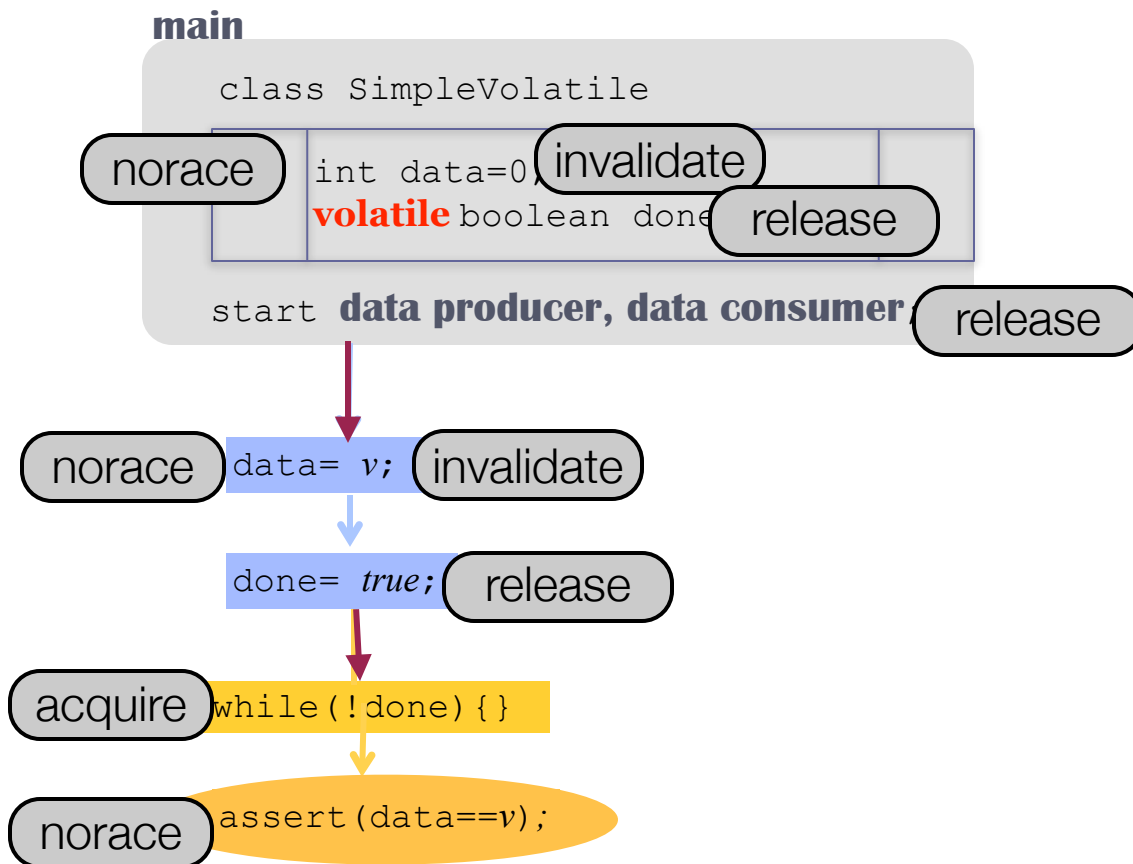
- Data race detection using summary function  $h$  for `SimpleVolatile`.





# Data Race Detection Algorithm

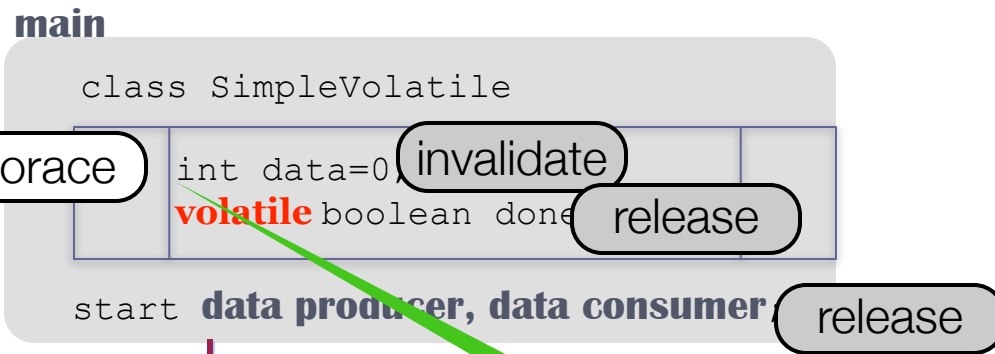
- Data race detection using summary function  $h$  for `SimpleVolatile`.



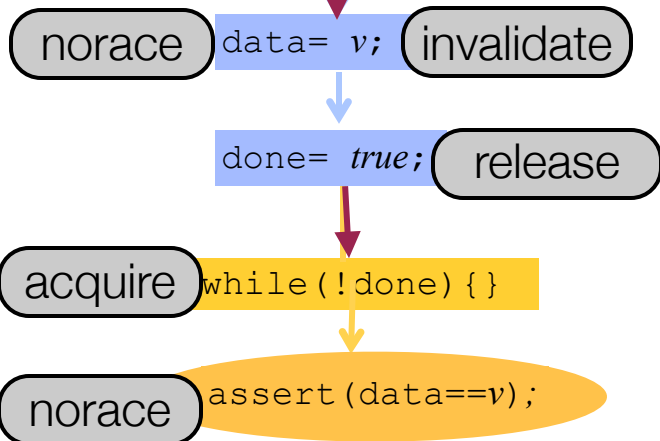
$h(\text{main})$	data
$h(\text{data producer})$	
$h(\text{data consumer})$	
$h(\text{done})$	

# Data Race Detection Algorithm

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



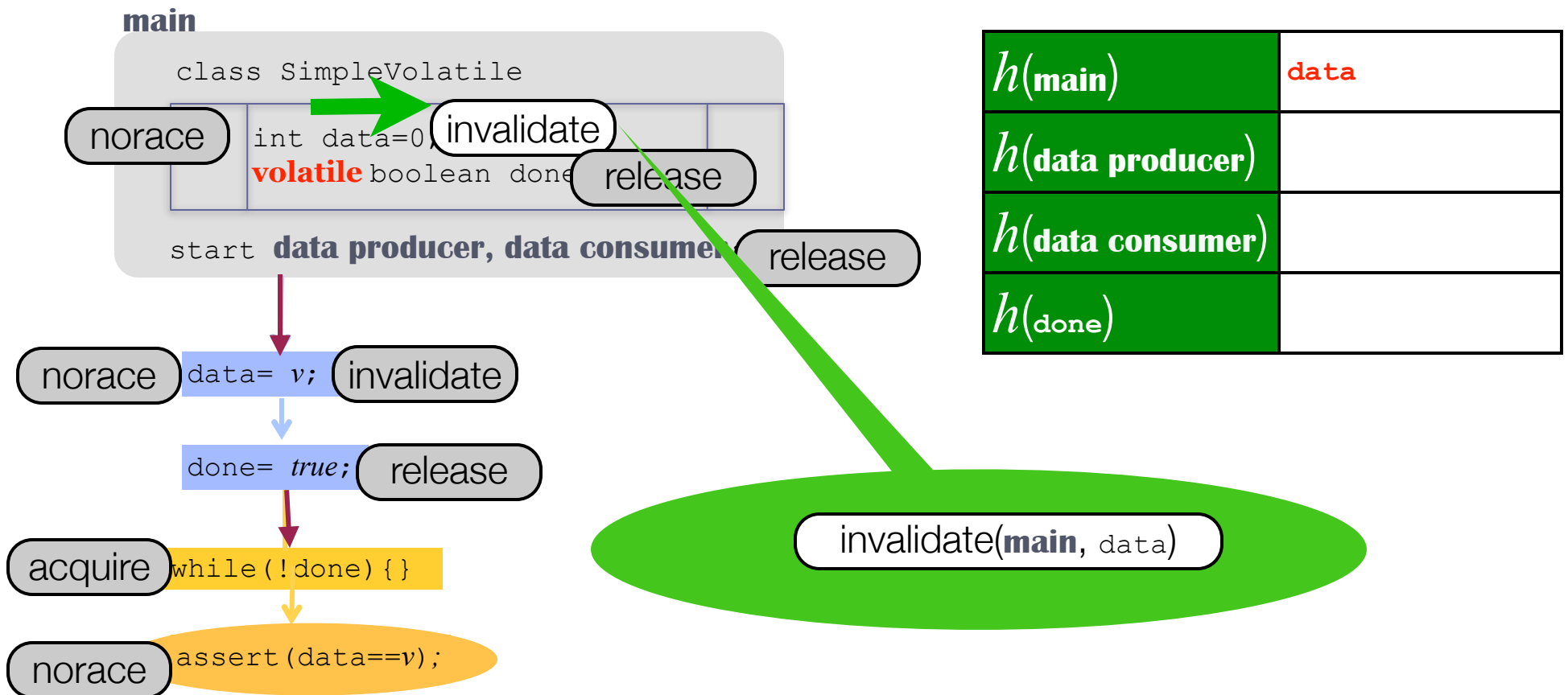
$h(\mathbf{main})$	<code>data</code>
$h(\mathbf{data\ producer})$	
$h(\mathbf{data\ consumer})$	
$h(\mathbf{done})$	



`norace(data, main)`

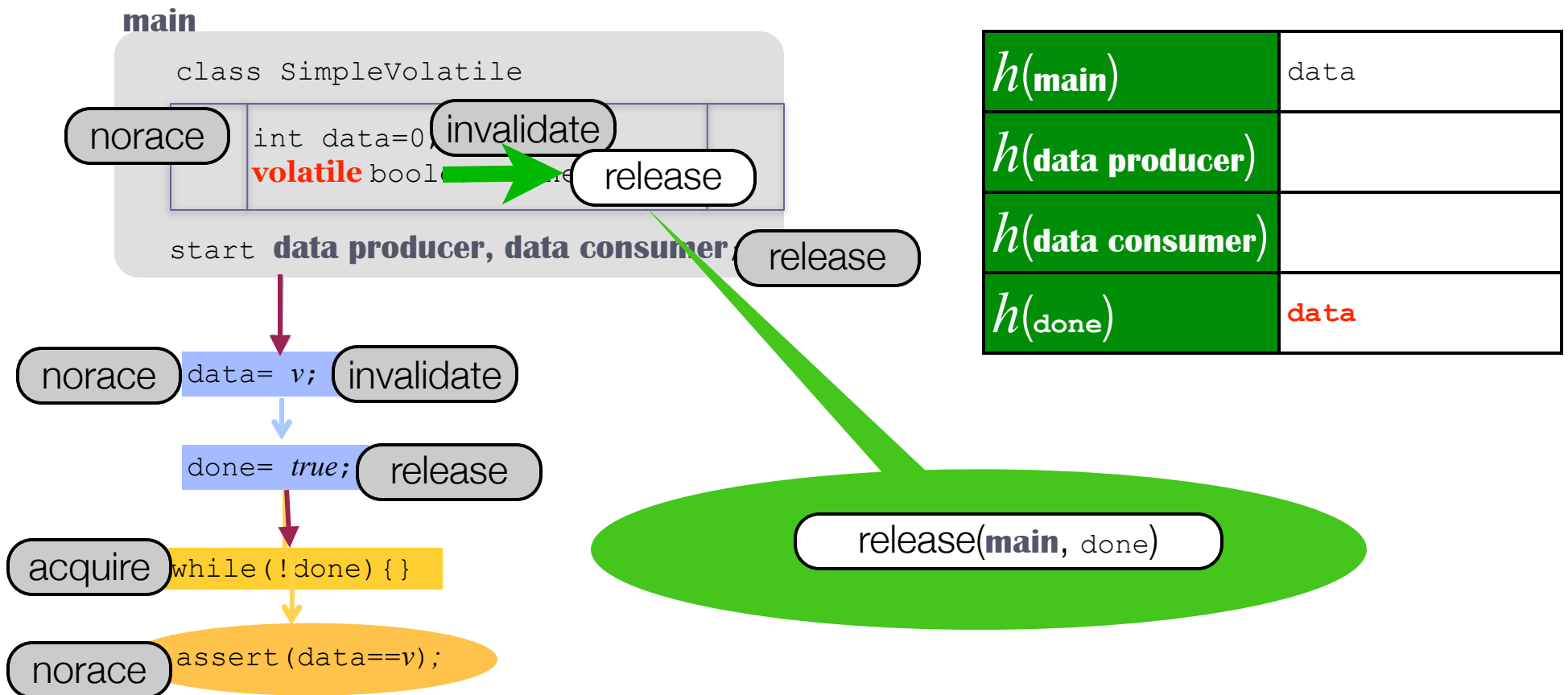
# Data Race Detection Algorithm

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



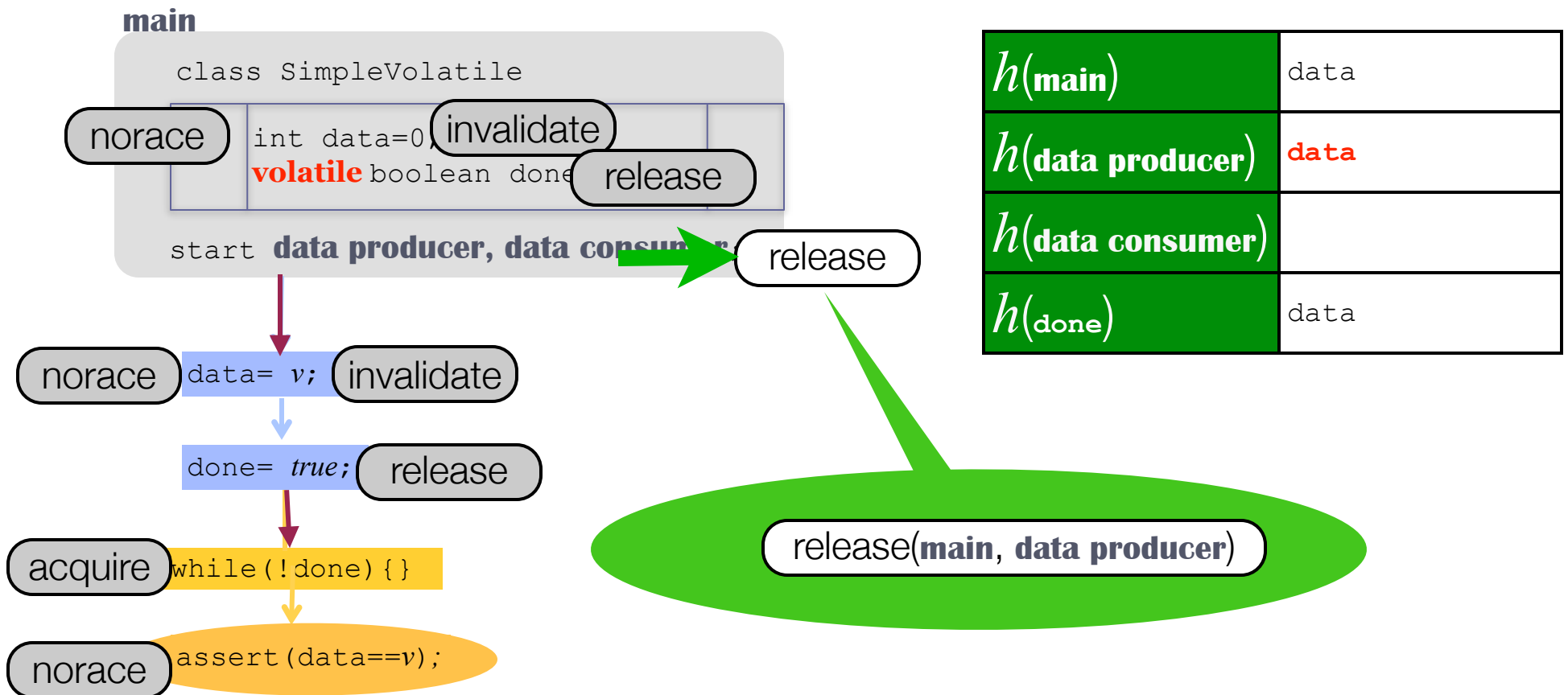
# Data Race Detection Algorithm

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



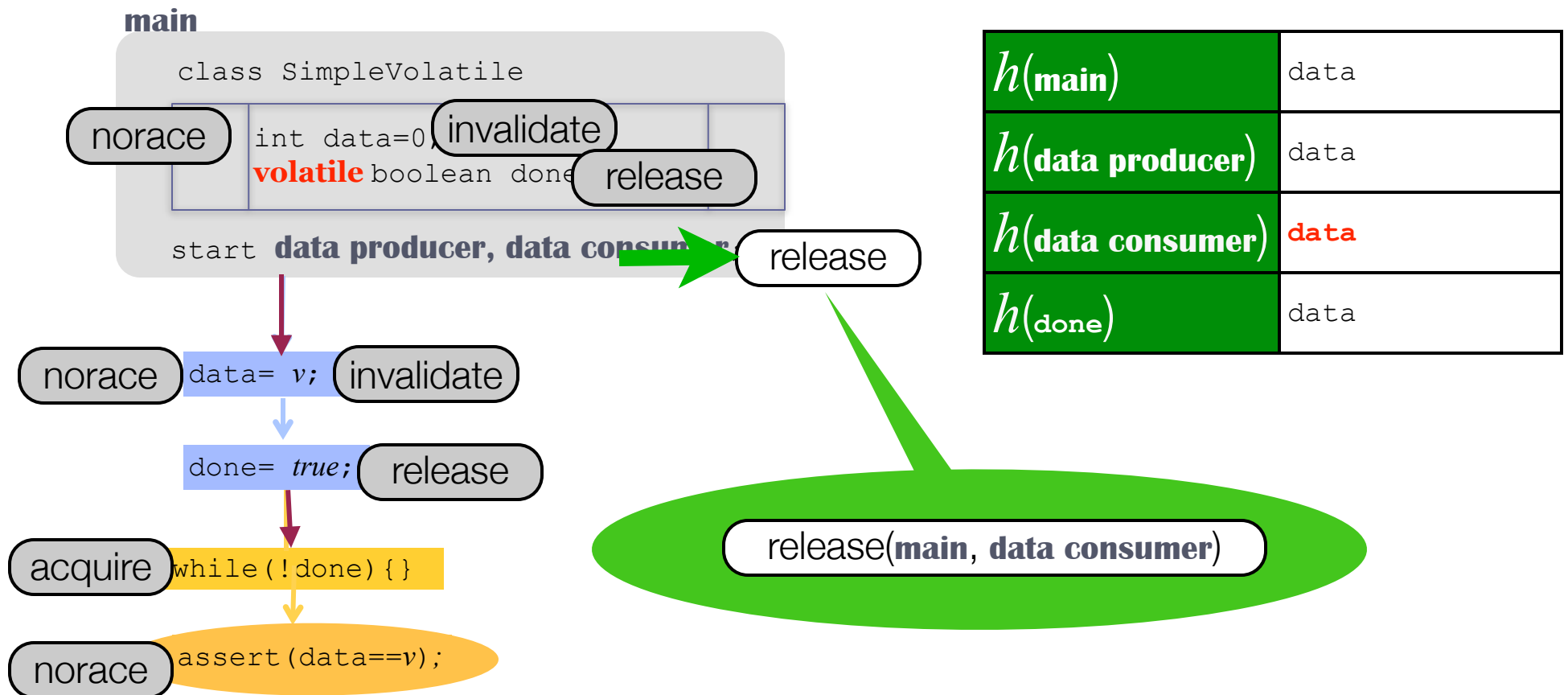
# Data Race Detection Algorithm

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



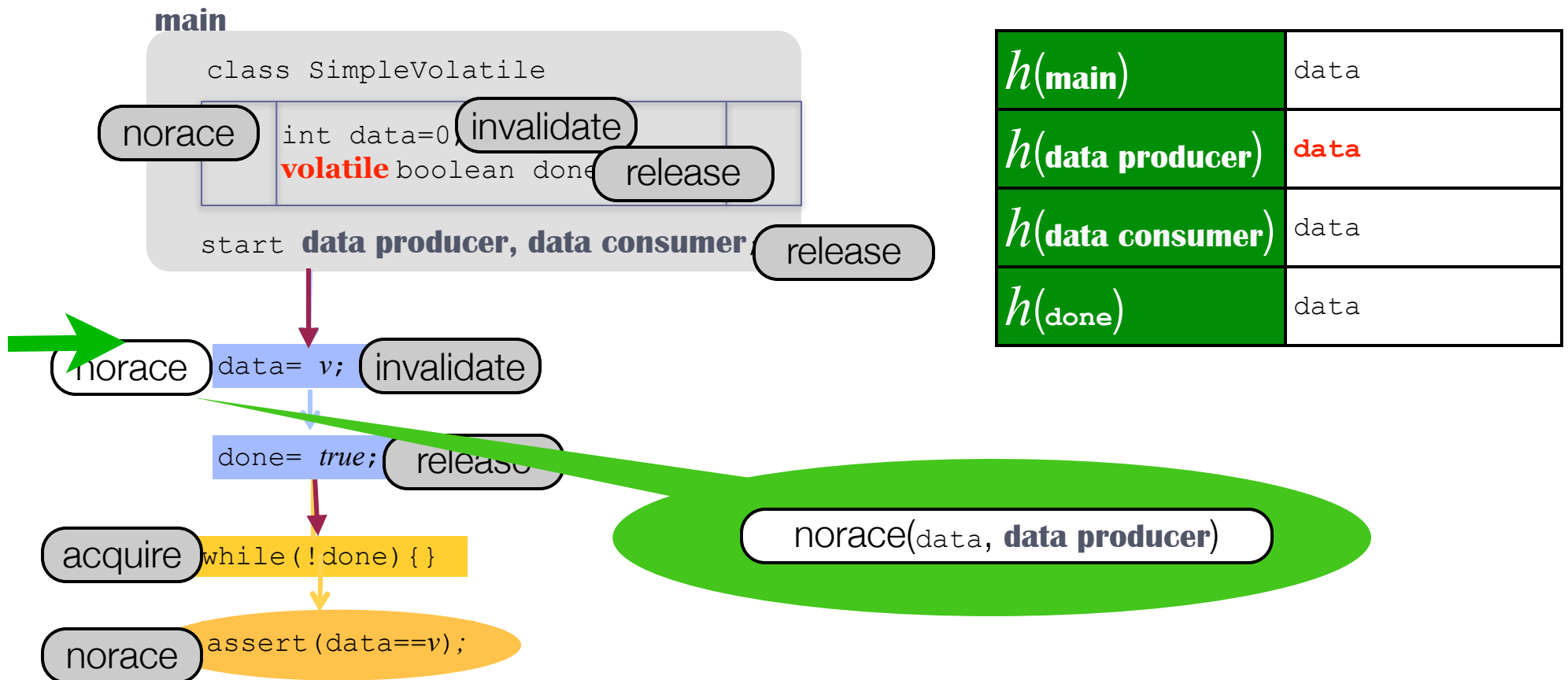
# Data Race Detection Algorithm

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



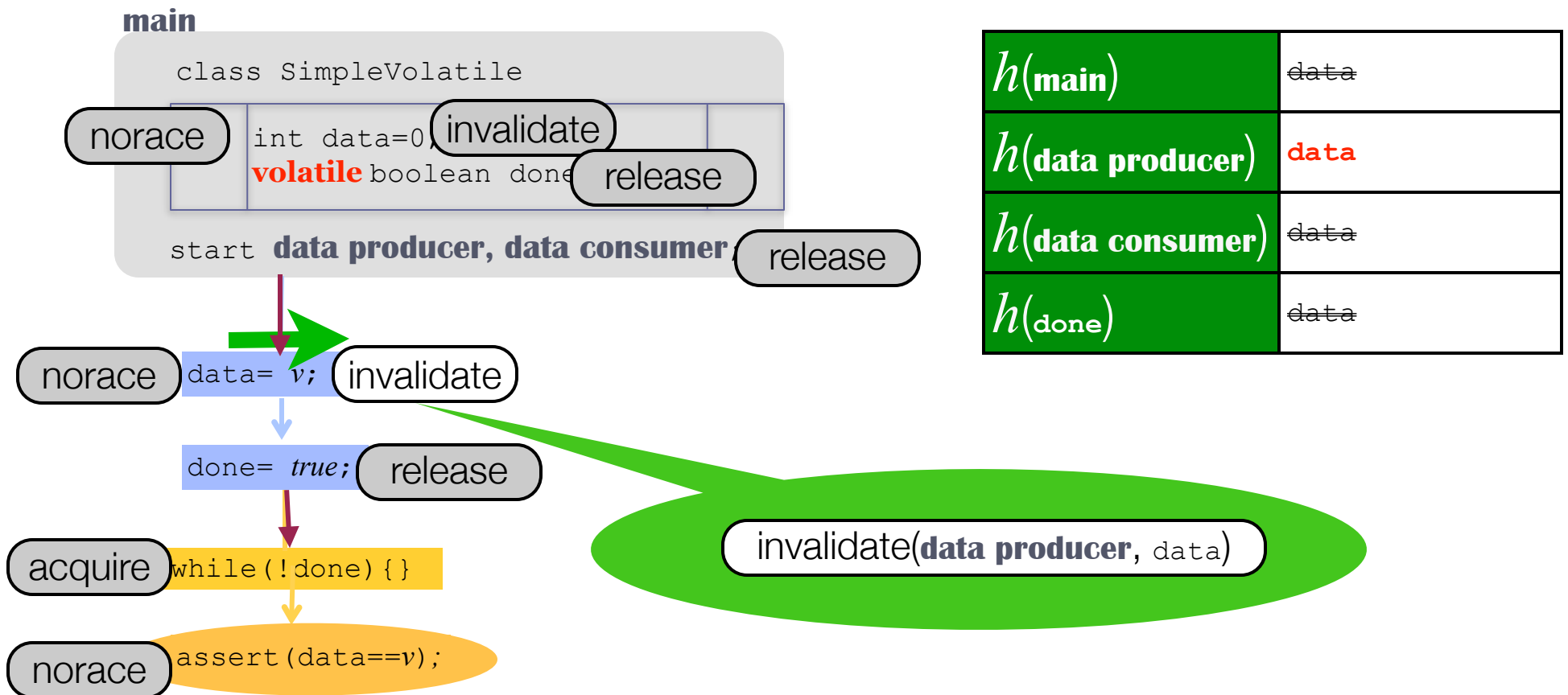
# Data Race Detection Algorithm

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



# Data Race Detection Algorithm

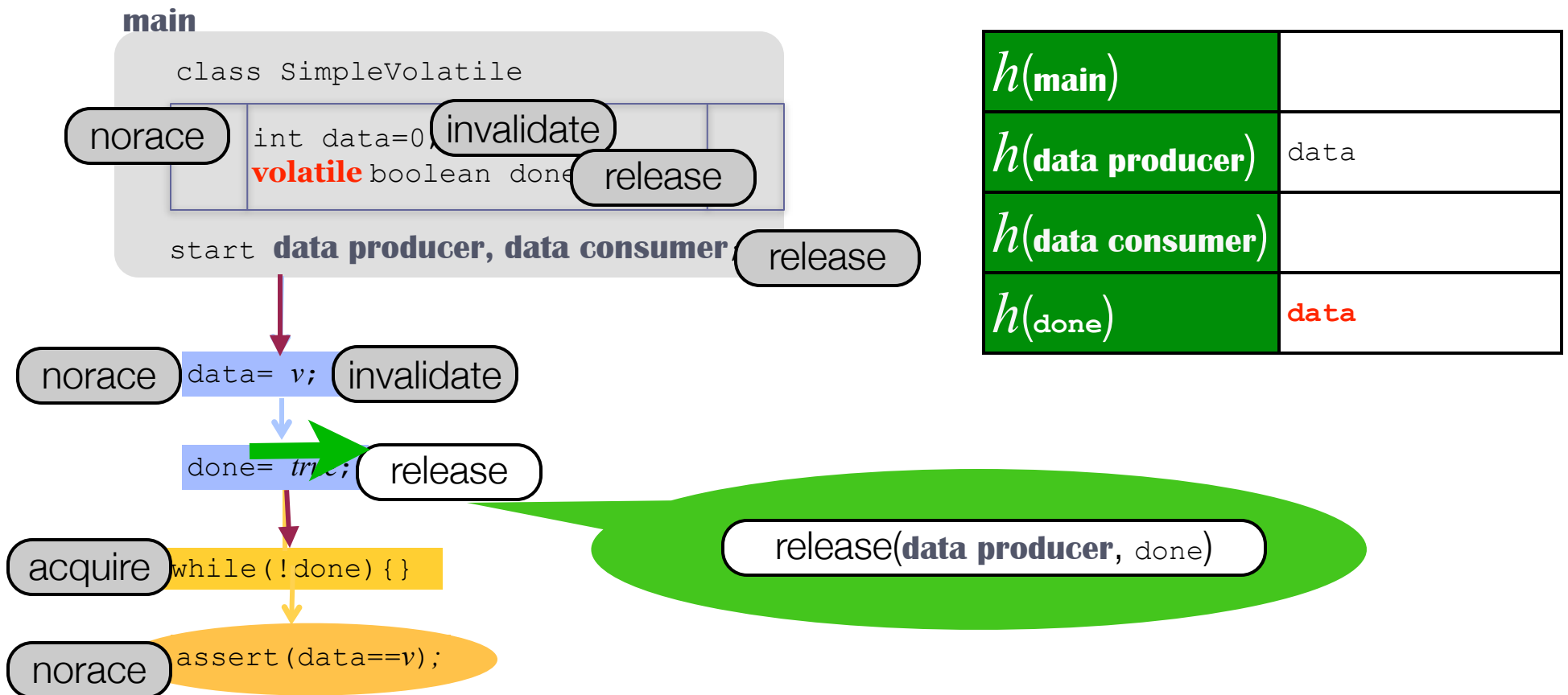
- Data race detection using summary function  $h$  for `SimpleVolatile`.





# 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`.

**main**

```
class SimpleVolatile
```

norace

```
int data=0;
```

invalidate

```
volatile boolean done;
```

release

```
start data producer, data consumer
```

release

$h(\text{main})$

$h(\text{data producer})$

data

$h(\text{data consumer})$

**data**

$h(\text{done})$

data

norace

```
data = v;
```

invalidate

```
done = true;
```

release

acquire(data consumer, done)

acquire

```
while (!done) {}
```

norace

```
assert (data == v);
```

# Data Race Detection Algorithm

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

**main**

```
class SimpleVolatile
```

norace

```
int data=0;
```

invalidate

```
volatile boolean done;
```

release

```
start data producer, data consumer
```

release

$h(\text{main})$

$h(\text{data producer})$

data

$h(\text{data consumer})$

data

$h(\text{done})$

data

norace

```
data = v;
```

invalidate

```
done = true;
```

release

acquire(data consumer, done)

acquire

```
while (!done) {}
```

norace

```
assert (data == v);
```

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`.

**main**

```
class SimpleVolatile
```

norace

```
int data=0;
```

invalidate

```
volatile boolean done;
```

release

```
start data producer, data consumer
```

release

$h(\text{main})$

$h(\text{data producer})$

data

$h(\text{data consumer})$

**data**

$h(\text{done})$

data

norace

```
data = v;
```

invalidate

```
done = true;
```

release

acquire

```
while (!done) {}
```

norace

```
assert (data == v);
```

norace(**data consumer**, data)

**OK!**



Data Race Analysis

# Experimental Result

---

# 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
<b>Total</b>	<b>39</b>	<b>132</b>

# 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

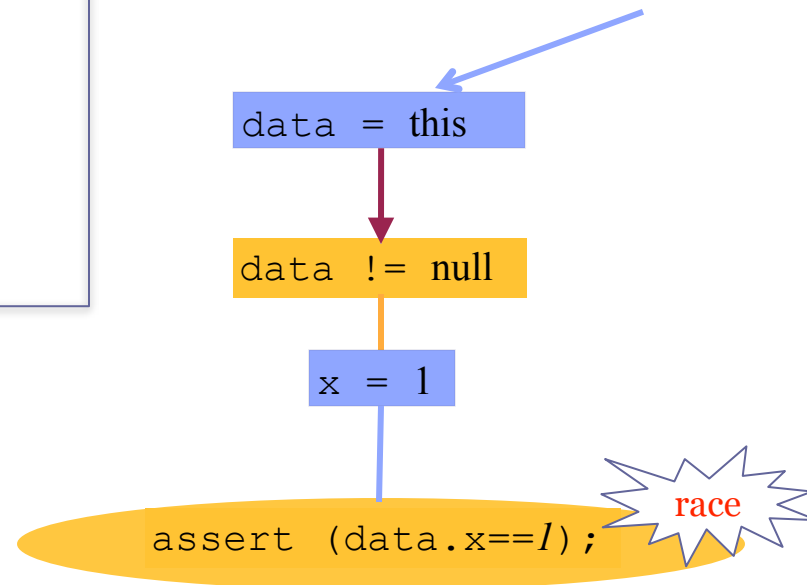
```
static volatile Data data;  
static class Data {  
    int x;  
    Data()  
    {  
        data = this;  
        x = 1;  
    }  
}
```

**Thread1**

```
new Data();
```

**Thread2**

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



# Experimental Result

---

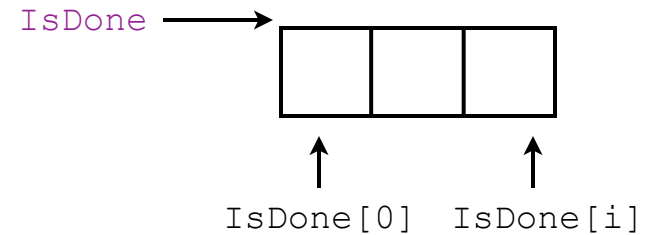
- Types of races
  - ➔ missing synchronization
  - ➔ unsafe publication
  - ➔ using volatile array instead of *java.util.concurrent.atomic* package
  - ➔ missing volatile declaration
  - ➔ benign race

# 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

---

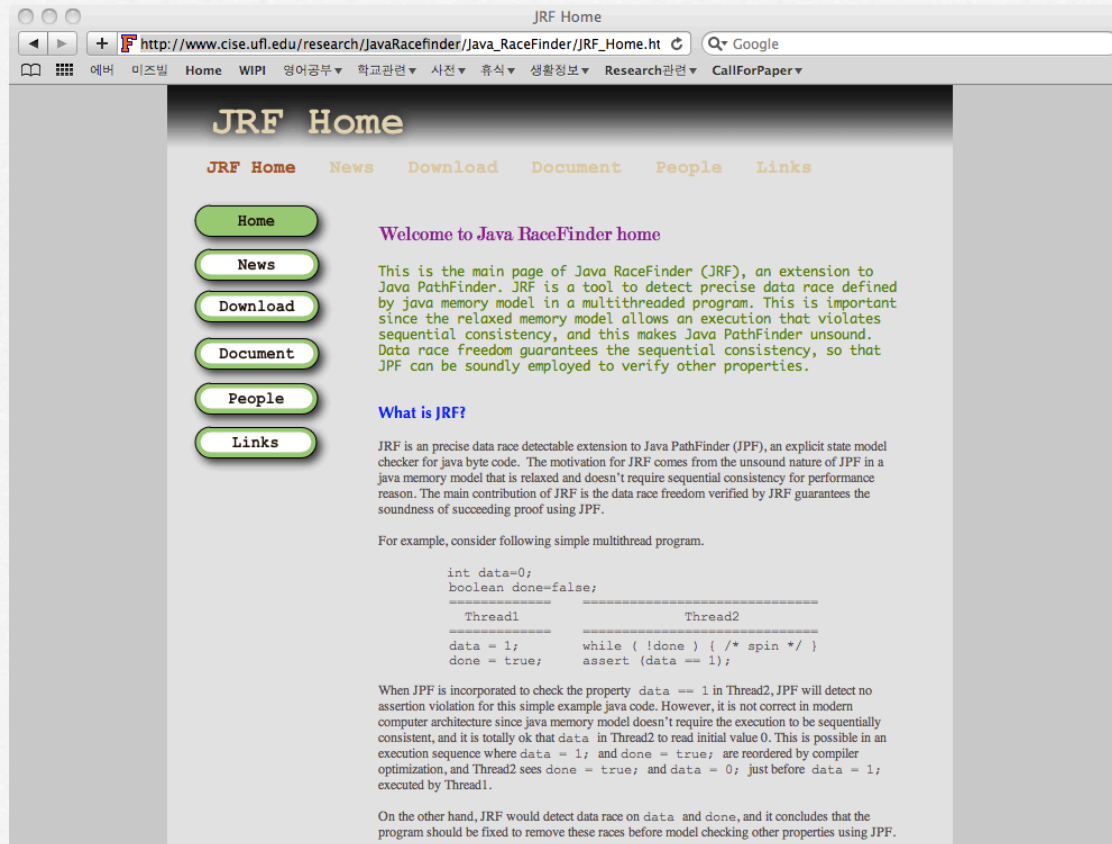
- JRF-E (JRF-Eliminator)  
: suggests how to eliminate found races
- JRF-M (JRF-Modular)  
: modular race detection using symbolic execution

---

Thank you!

# JRF version 1.0

□ <http://www.cise.ufl.edu/research/JavaRacefinder/>



The screenshot shows a web browser window displaying the JRF Home page. The browser's address bar shows the URL [http://www.cise.ufl.edu/research/JavaRacefinder/JRF\\_Home.ht](http://www.cise.ufl.edu/research/JavaRacefinder/JRF_Home.ht). The page title is "JRF Home". The main content area features a navigation menu with buttons for Home, News, Download, Document, People, and Links. The "Home" button is highlighted. Below the navigation menu, the page displays a welcome message: "Welcome to Java RaceFinder home". This is followed by a paragraph explaining that JRF is an extension to Java PathFinder (JPF) designed to detect precise data races in multithreaded programs. A section titled "What is JRF?" provides a detailed explanation of the tool's purpose and how it addresses the limitations of JPF. A code example is provided to illustrate a data race scenario involving two threads, Thread1 and Thread2, which access a shared variable 'data'. The code shows Thread1 setting 'data' to 1 and Thread2 asserting that 'data' is 1, which can be violated due to compiler optimizations. The text concludes by stating that JRF would detect such a race and recommend fixing the program before using JPF for model checking.

JRF Home

[JRF Home](#) [News](#) [Download](#) [Document](#) [People](#) [Links](#)

Home

News

Download

Document

People

Links

Welcome to Java RaceFinder home

This is the main page of Java RaceFinder (JRF), an extension to Java PathFinder. JRF is a tool to detect precise data race defined by java memory model in a multithreaded program. This is important since the relaxed memory model allows an execution that violates sequential consistency, and this makes Java PathFinder unsound. Data race freedom guarantees the sequential consistency, so that JPF can be soundly employed to verify other properties.

What is JRF?

JRF is an precise data race detectable extension to Java PathFinder (JPF), an explicit state model checker for java byte code. The motivation for JRF comes from the unsound nature of JPF in a java memory model that is relaxed and doesn't require sequential consistency for performance reason. The main contribution of JRF is the data race freedom verified by JRF guarantees the soundness of succeeding proof using JPF.

For example, consider following simple multithread program.

```
int data=0;
boolean done=false;
-----
Thread1      Thread2
-----
data = 1;    while ( !done ) { /* spin */ }
done = true; assert (data == 1);
```

When JPF is incorporated to check the property `data == 1` in Thread2, JPF will detect no assertion violation for this simple example java code. However, it is not correct in modern computer architecture since java memory model doesn't require the execution to be sequentially consistent, and it is totally ok that `data` in Thread2 to read initial value 0. This is possible in an execution sequence where `data = 1;` and `done = true;` are reordered by compiler optimization, and Thread2 sees `done = true;` and `data = 0;` just before `data = 1;` executed by Thread1.

On the other hand, JRF would detect data race on `data` and `done`, and it concludes that the program should be fixed to remove these races before model checking other properties using JPF.



# 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 history 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

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

# Extending JPF

- Unsafe Publication

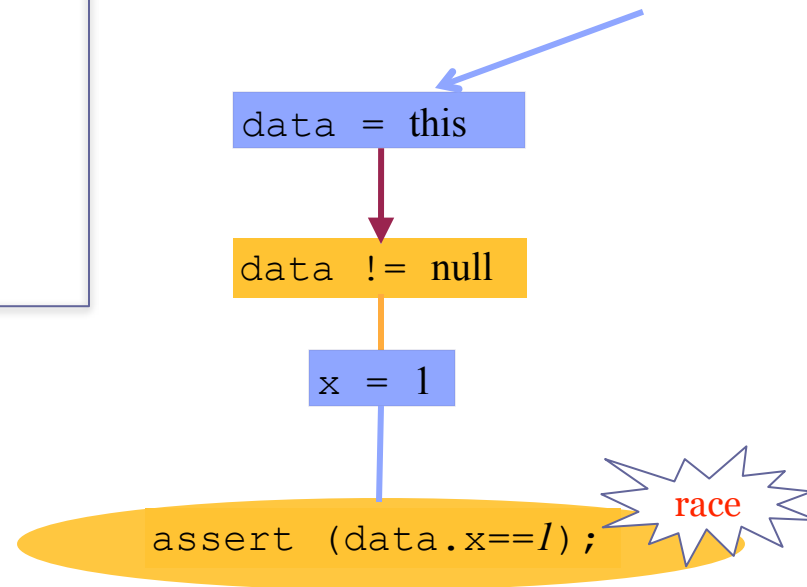
```
static volatile Data data;  
static class Data {  
    int x;  
    Data()  
    {  
        data = this;  
        x = 1;  
    }  
}
```

### 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);
```

PreciseRaceDetector  
find a race on **data**  
but not on **x**

x = 1

assert (data.x==1);

race