Concurrent Programming: JR Language

Michał Pałka

Chalmers University of Technology
Gothenburg, Sweden

September 24, 2012

JR

- JR — academic programming language for concurrency
- Extension of Java
- Advantage: Adds many expressive message passing primitives
- Disadvantage: Java is already complicated, JR is even more
- Lab 3 is based on JR

Hello, World!

```
import edu.ucdavis.jr.JR;

public class Hello {
    public static void main (String[] args) {
        System.out.println ("Hello, world!");
    }
};
```

Hello, World!

```
import edu.ucdavis.jr.JR;

public class Hello {
    public static void main (String[] args) {
        System.out.println ("Hello, world!");
    }
};
```

Imports JR functions

Hello, World!

```
import edu.ucdavis.jr.JR;
```

Imports JR functions

```
public class Hello {
    public static void main (String[] args) {
        System.out.println ("Hello, world!");
    }
};
```

Save to `Hello.jr`

```
$ jr Hello
Hello, world!
```

Compilation issues

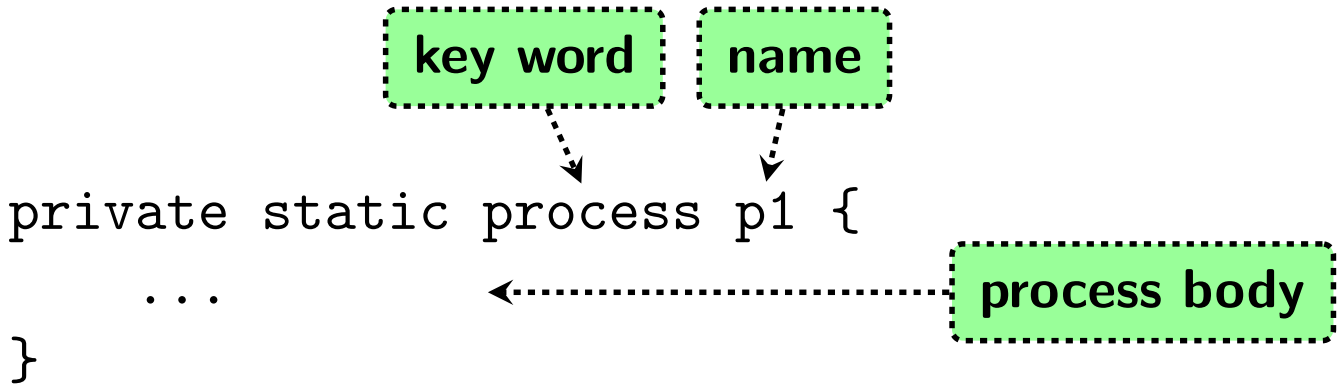
- JR compiles all `*.jr` files in your directory.
- Their contents must match their file names.

Processes

```
private static process p1 {
    ...
}
```

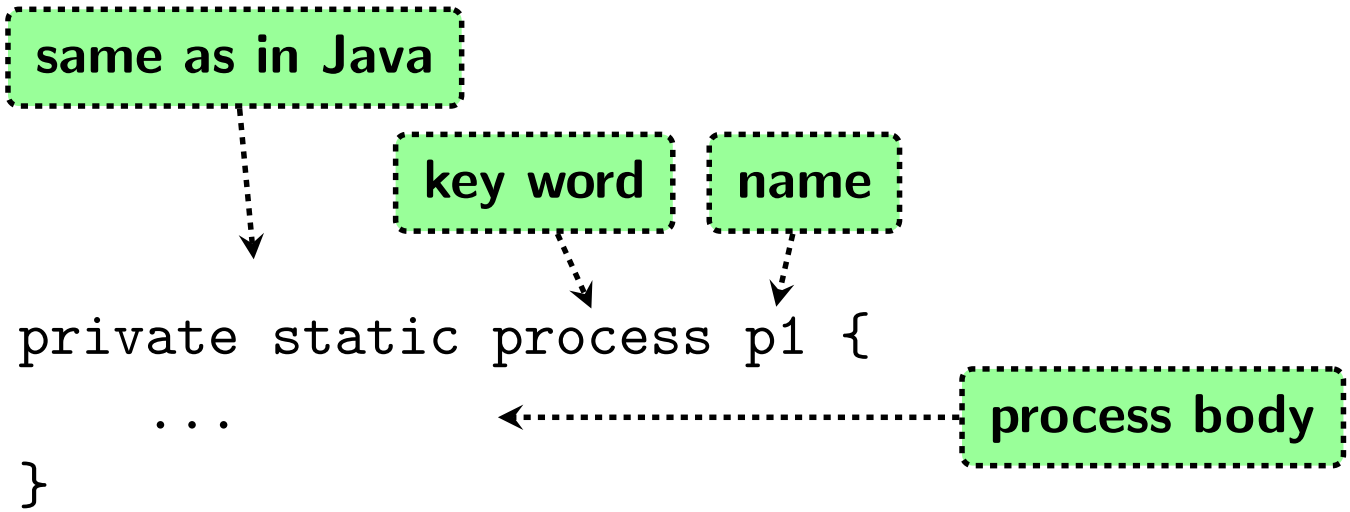
- Process that runs concurrently to everything else.

Processes



- Process that runs concurrently to everything else.

Processes

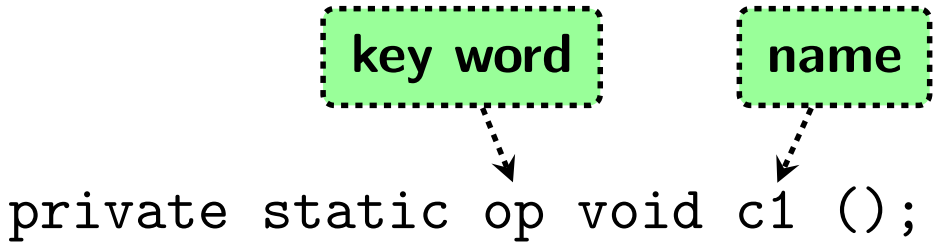


- Process that runs concurrently to everything else.

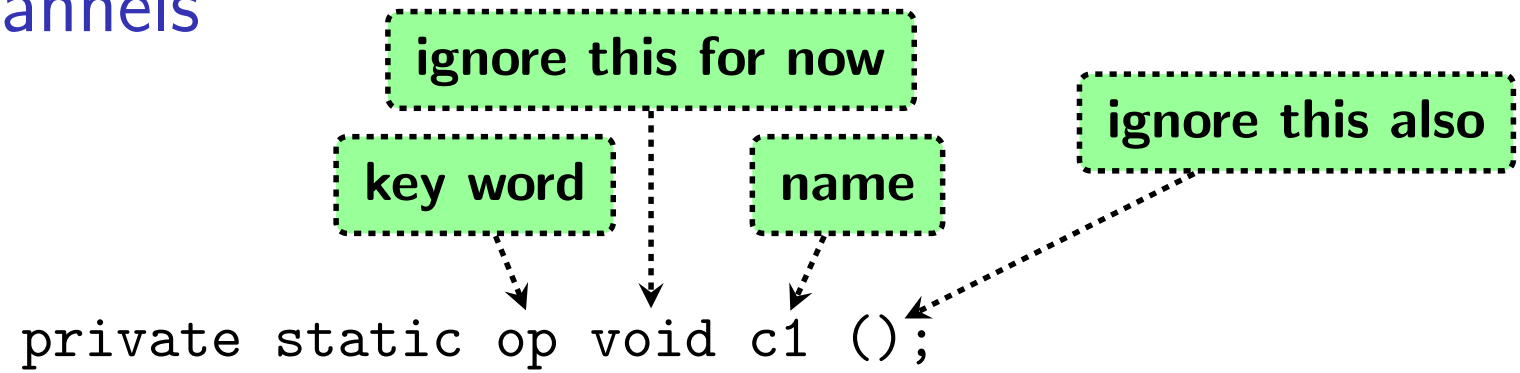
Channels

```
private static op void c1 ();
```

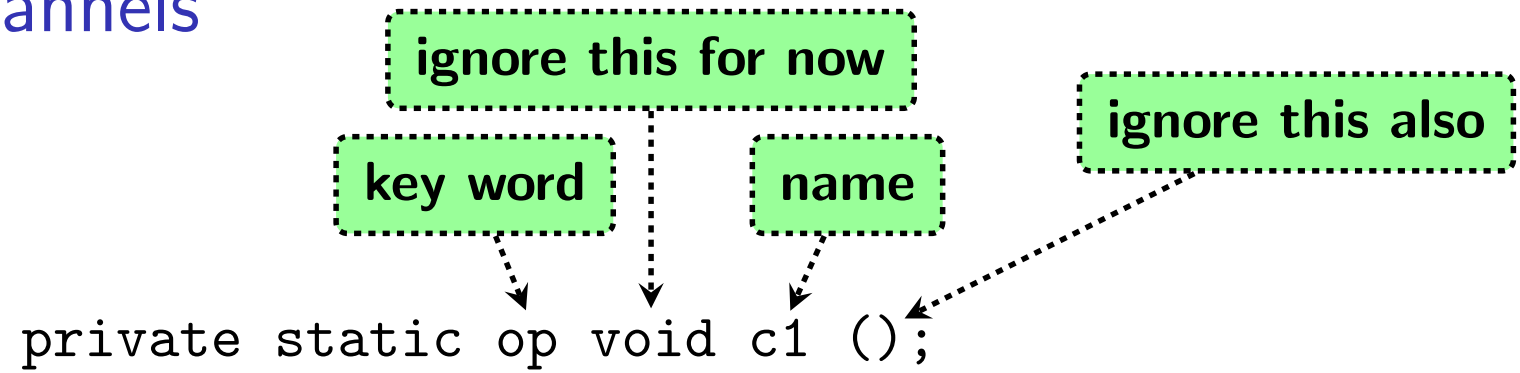
Channels



Channels

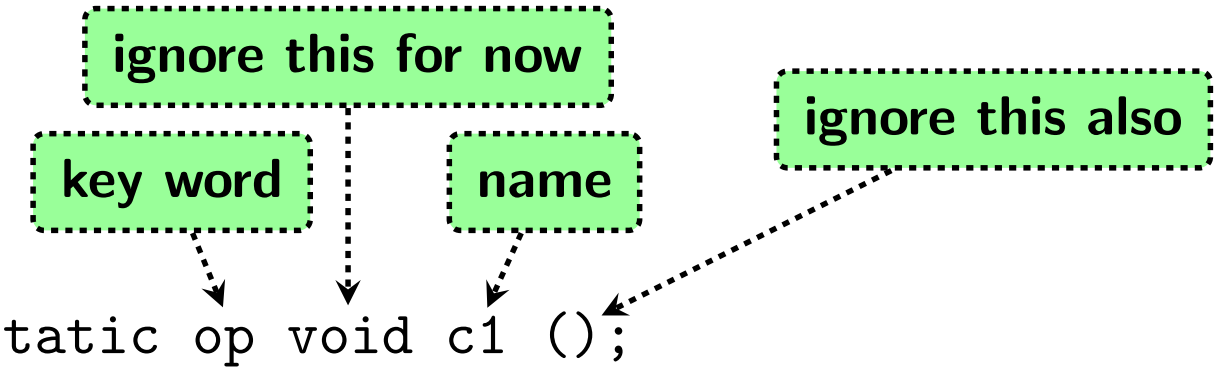

ignore this for now
key word
name
ignore this also
private static op void c1 ();

Channels



- Channel, which can be used to send and receive messages.
- Many processes can send and receive on the same channel.
- Messages sent to a channel are queued.

Channels



- Channel, which can be used to send and receive messages.
- Many processes can send and receive on the same channel.
- Messages sent to a channel are queued.

Sending and receiving:

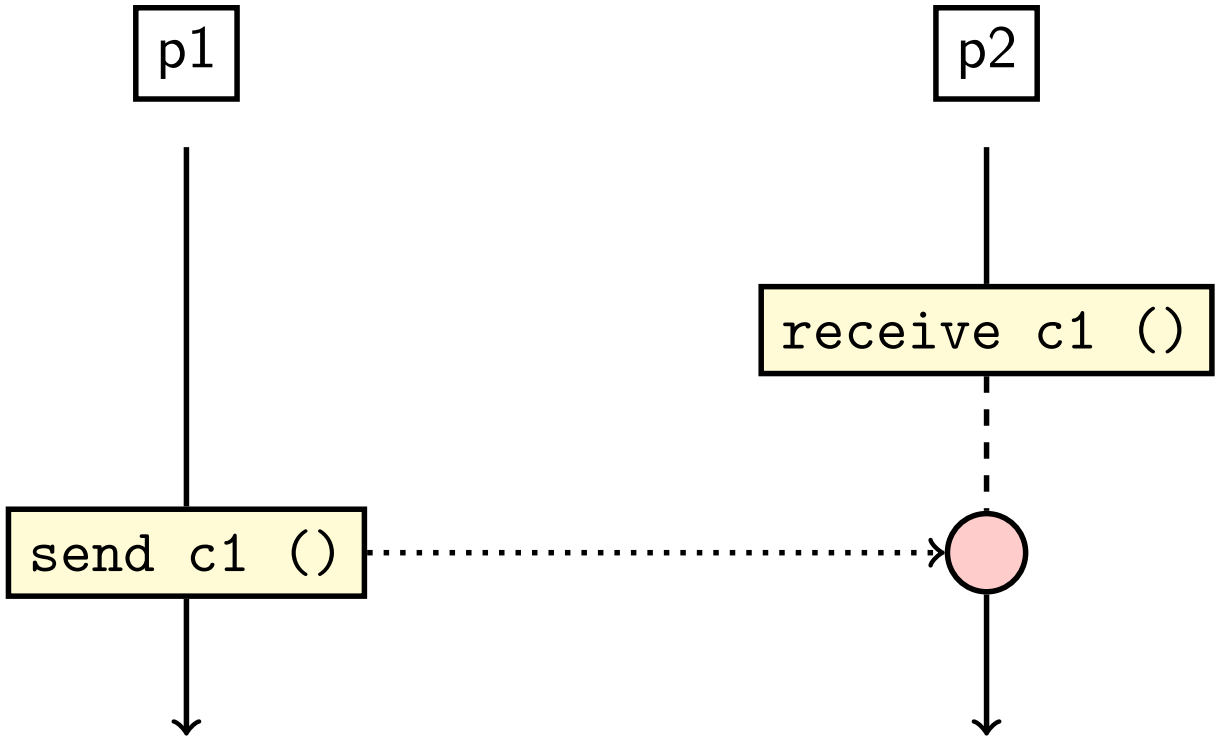
```
send c1 ();
```

```
receive c1 ();
```

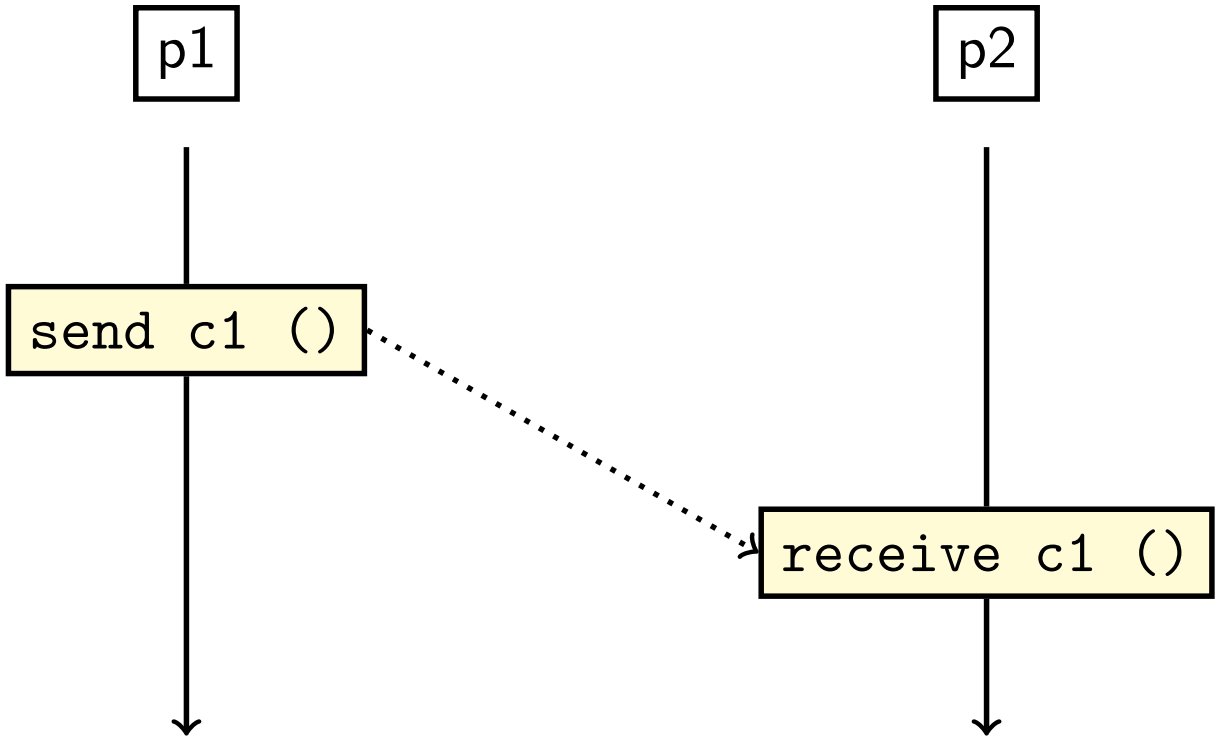
By the way

In JR use `JR.nap()` instead of `Thread.sleep()`.

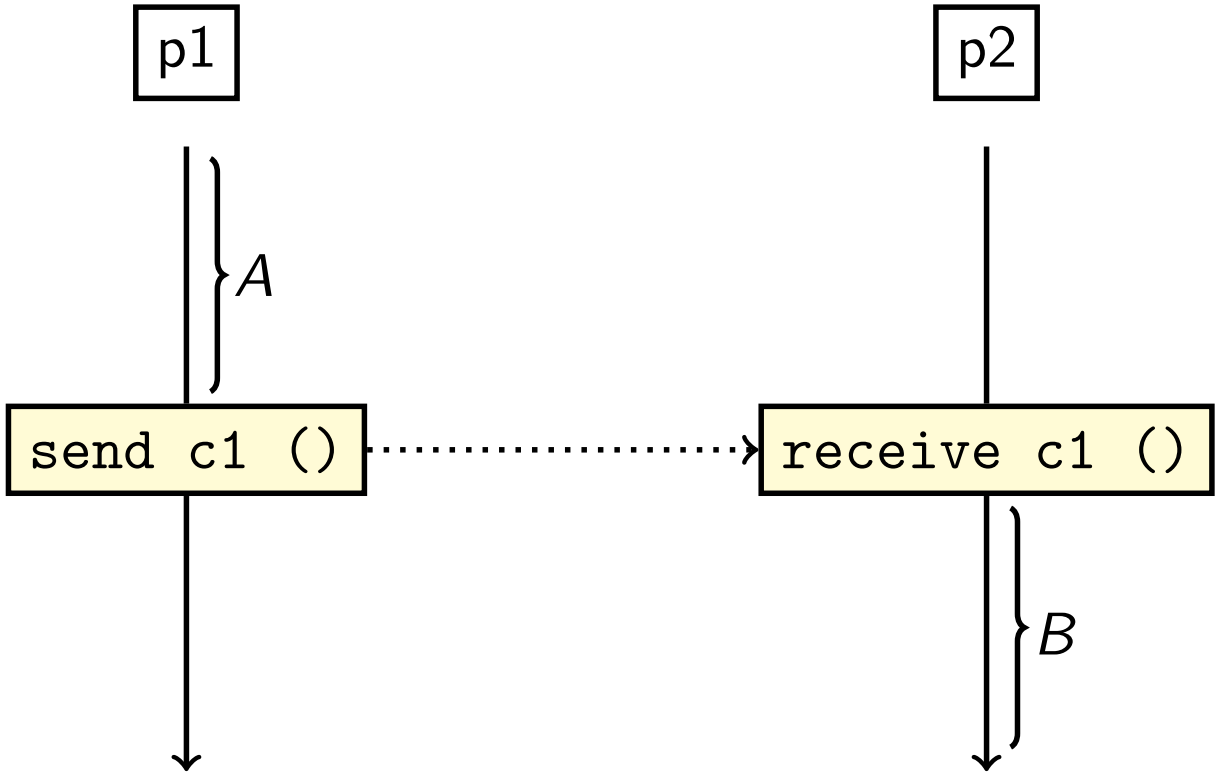
Message send



Message send (cont.)



Message send (cont.)



- Send and receive ensure that actions in A are executed before actions in B.

And one more thing

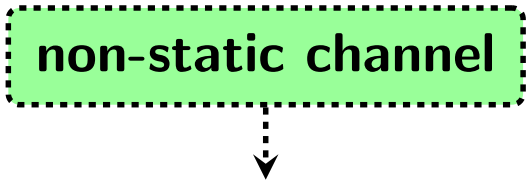
And one more thing

- JR has deadlock detection.
- When deadlock occurs, your program will exit.

Summary

- Hello, compilation (and issues)
- Channels
- Sending end receiving messages
- Deadlock detection

Static, non-static




```
private op void c1 ();


public static void main (String[] args) {
    send c1 ();
}
```

Static, non-static

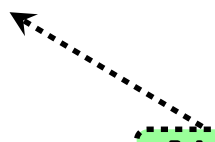
non-static channel

```
private op void c1 ();


public static void main (String[] args) {
    send c1 ();
}
```

non-static operation op void c1() cannot be referenced from a static context

Static, non-static (cont.)

```
public class Static {
    private op void c1 ();

    public static void main (String[] args) {
        Static s = new Static ();
        send s.c1 ();
    }
};
```

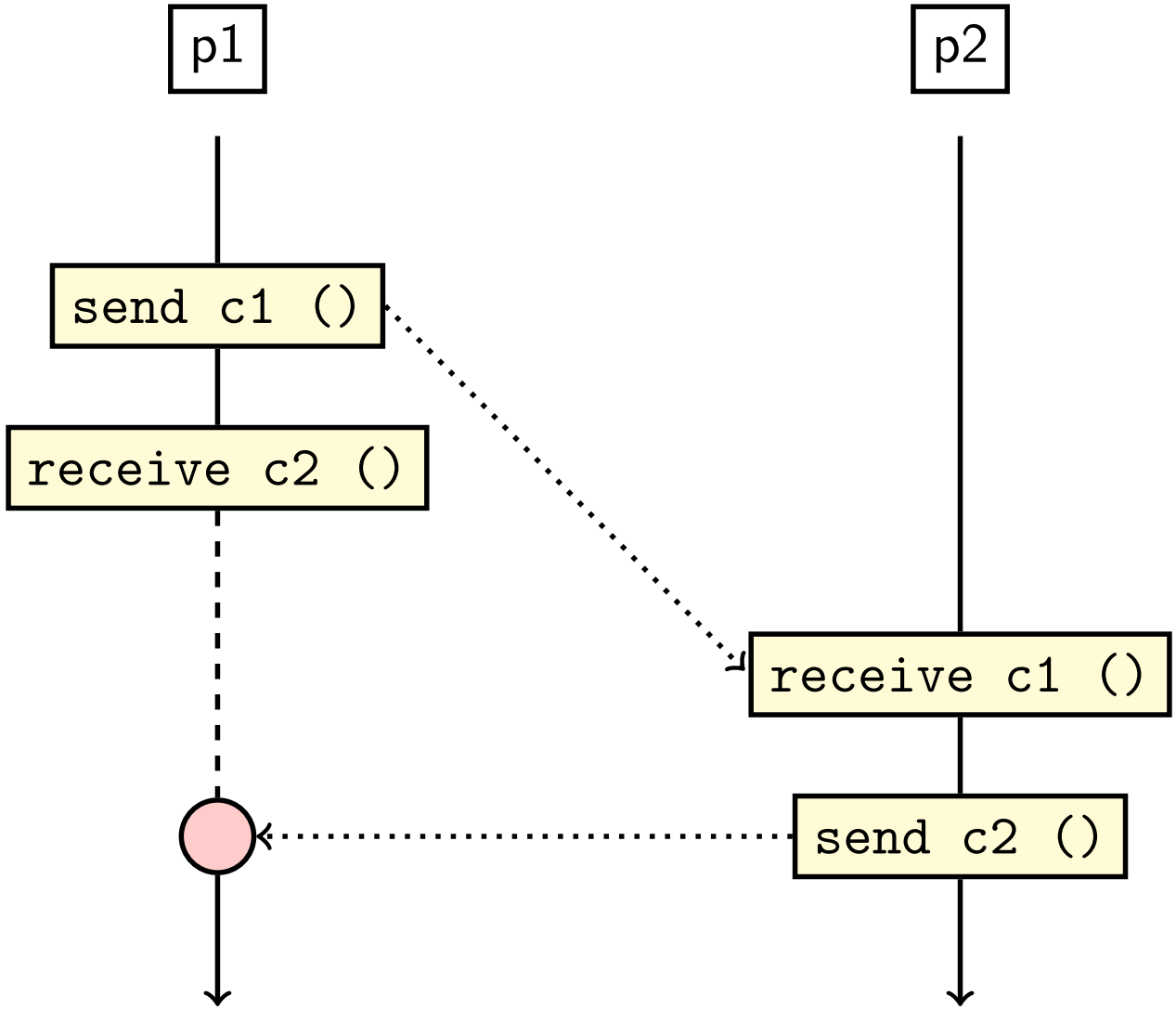


Non-static channels and processes are created together with an object.

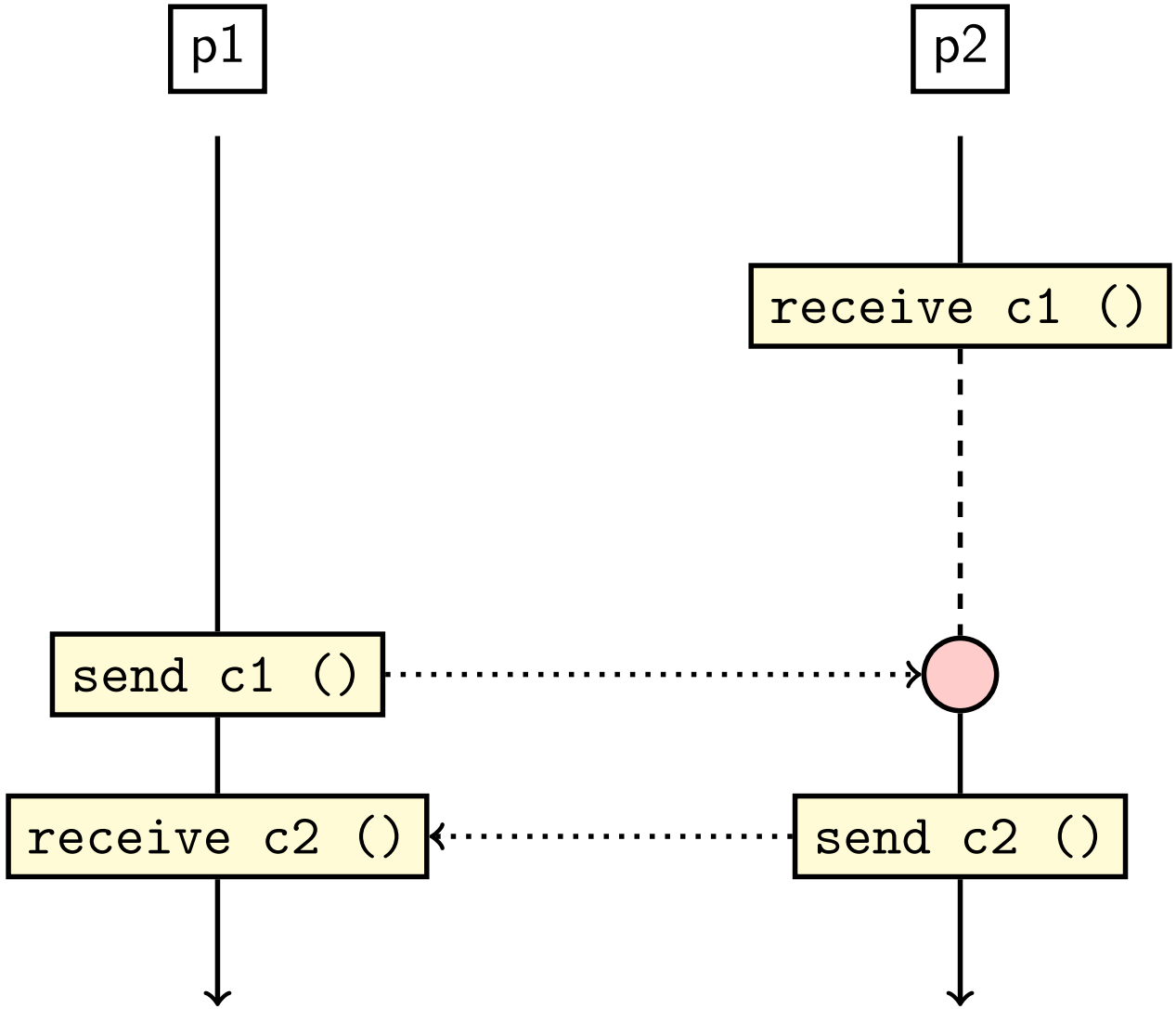
Rendez-vous

```
private static op void c1 ();
private static op void c2 ();

private static process p1 {
    // some code
    send c1 ();
    receive c2 ();
    // more code
}

private static process p2 {
    // some code
    receive c1 ();
    send c2 ();
    // more code
}
```

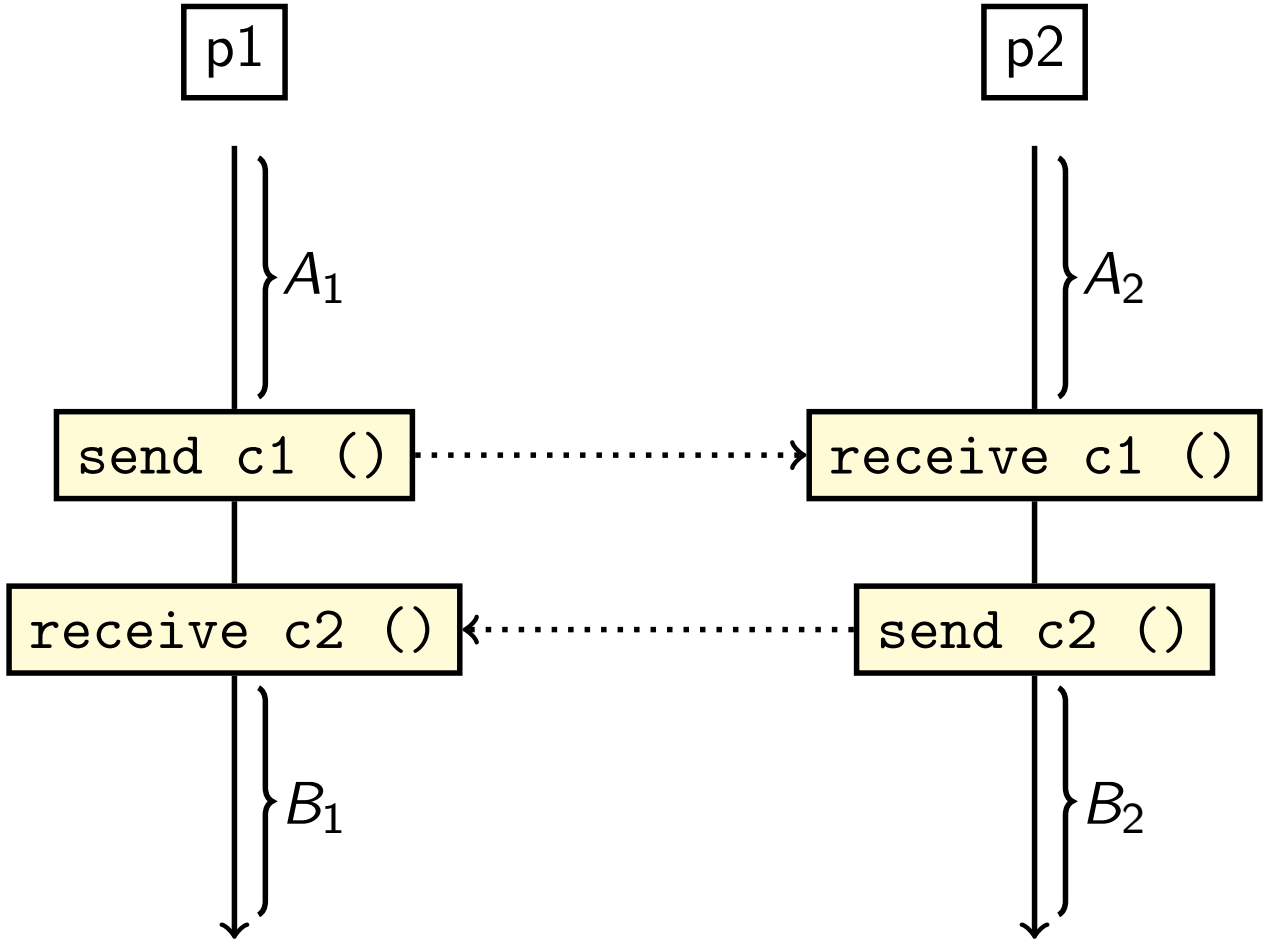
Rendez-vous (cont.)



Rendez-vous (cont.)



Rendez-vous (cont.)



- This pattern ensures that actions from A_1 occur before actions from B_2 and actions from A_2 occur before actions from B_1.

Rendez-vous (cont.)

- It is possible to implement rendez-vous (RDV) using asynchronous send and two channels.
- JR provides also direct support for rendez-vous.

Call

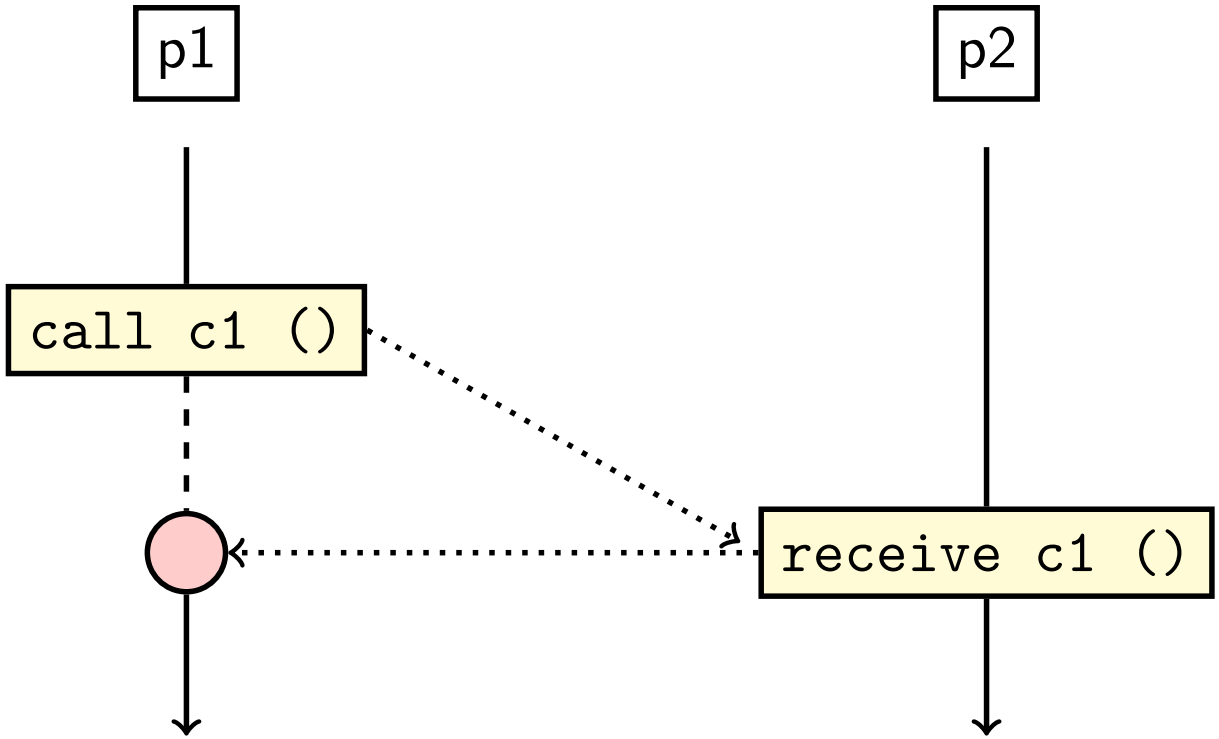
```
private static op void c1 ();

private static process p1 {
    // some code
    call c1 ();
    // more code
}

private static process p2 {
    // some code
    receive c1 ();
    // more code
}
```

Call

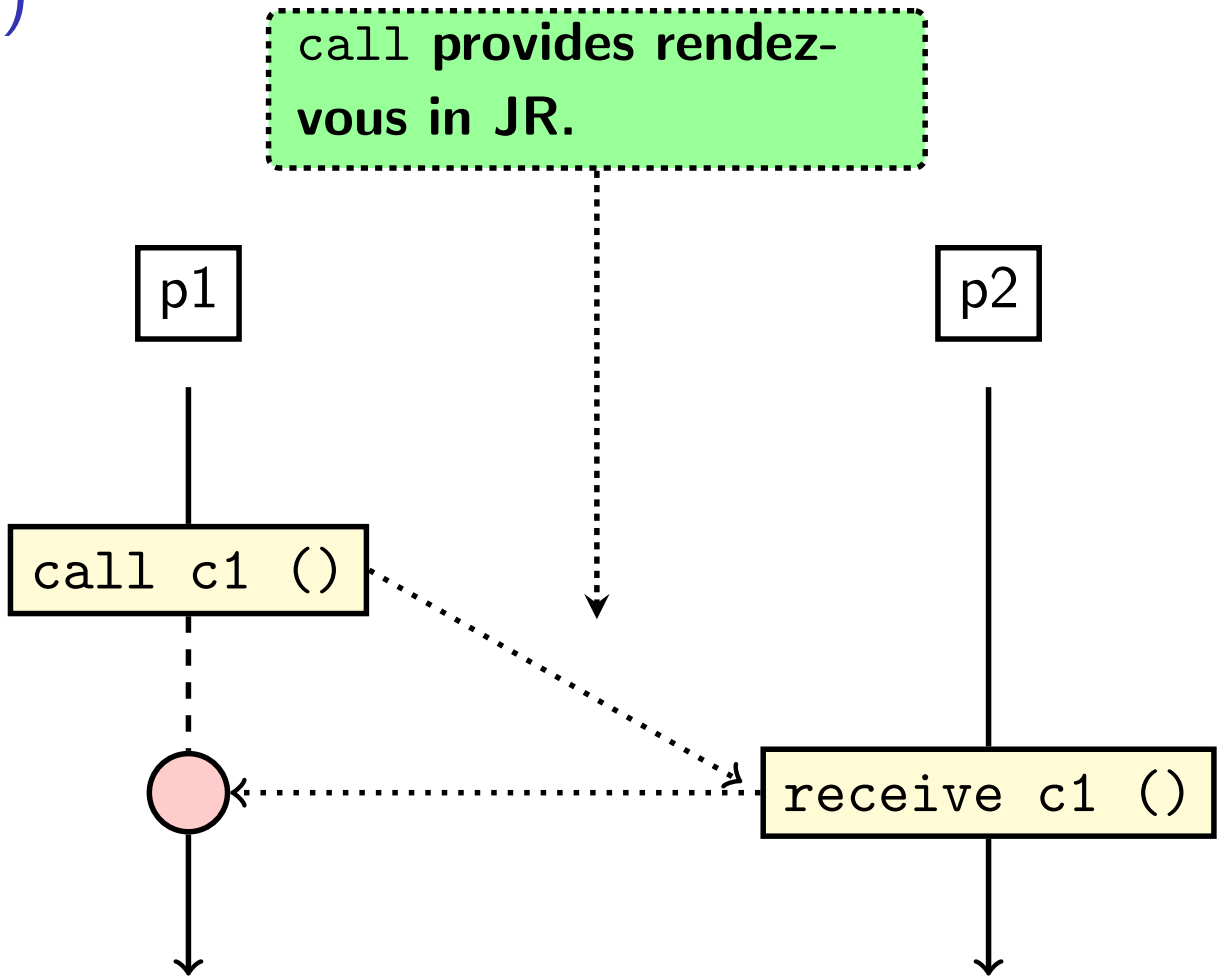
```
private static op void c1 ();

private static process p1 {
    // some code
    call c1 ();
    // more code
}

private static process p2 {
    // some code
    receive c1 ();
    // more code
}
```

`call` **will wait until the other process performs the receive.**

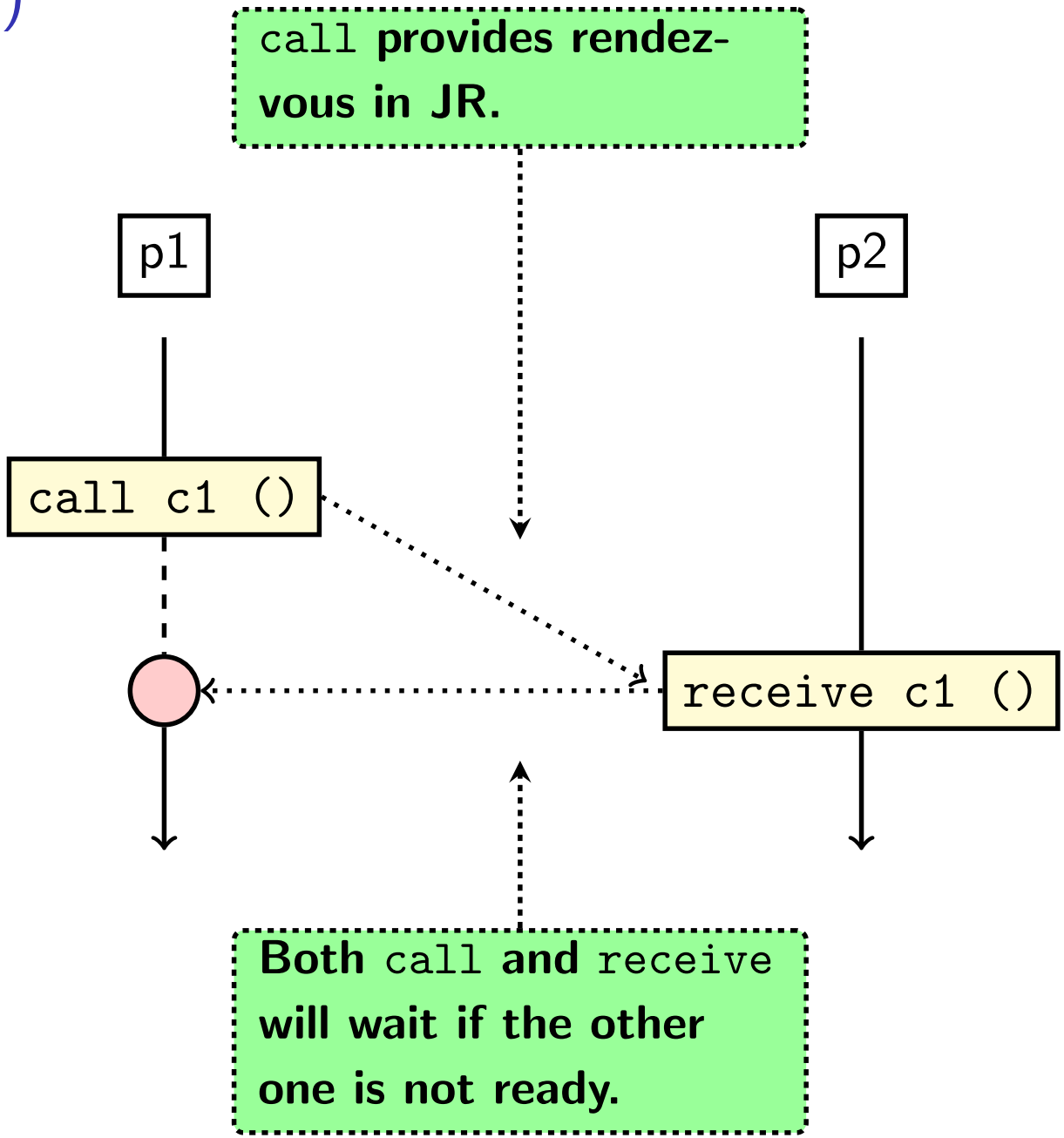
Call (cont.)



Call (cont.)



Call (cont.)


call provides rendez-vous in JR.
p1
p2
call c1 ()
receive c1 ()
Both call and receive will wait if the other one is not ready.

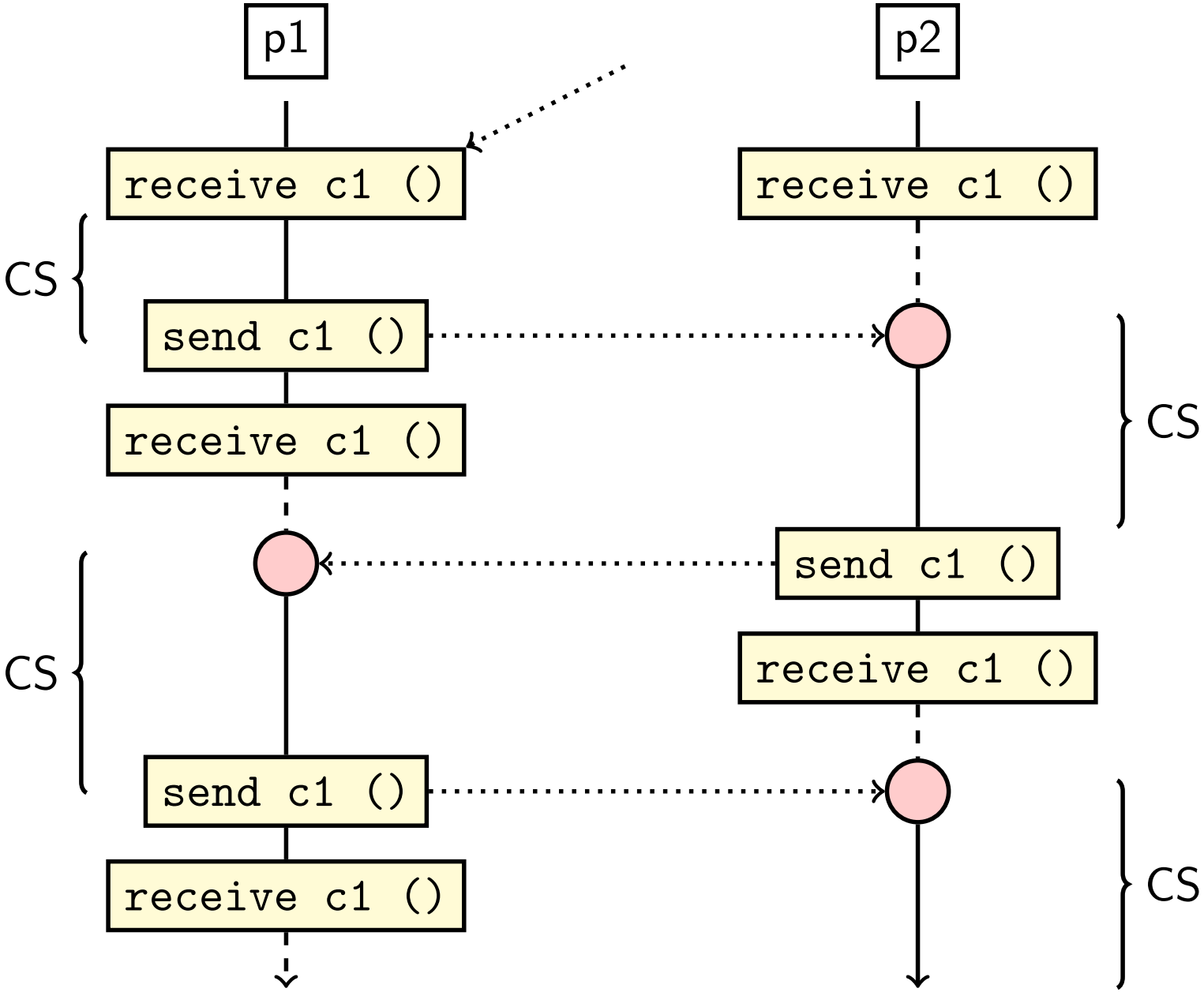
Summary

- Static/non-static channels (and processes)
- Rendez-vous using two messages
- The `call` statement (gives us RDV directly)

Puzzle

```
private static op void c1 ();

private static process p1 {
    for (int i = 0; i < 10; ++i) {
        receive c1 ();
        // Some code
        send c1 ();
    } }
private static process p2 {
    for (int i = 0; i < 10; ++i) {
        receive c1 ();
        // Some code
        send c1 ();
    } }
public static void main (String[] args) {
    send c1 ();
}
```

Puzzle (cont.)


p1
p2
receive c1 ()
receive c1 ()
CS
send c1 ()
receive c1 ()
CS
send c1 ()
receive c1 ()
CS
send c1 ()
receive c1 ()
CS

Semaphore notation

```
private static sem s1 = 1;

private static process p1 {
    for (int i = 0; i < 10; ++i) {
        P (s1);
        // Critical section
        V (s1);
    } }
```

Semaphore notation

Same as defining a channel and sending a message to it.

```
private static sem s1 = 1;

private static process p1 {
    for (int i = 0; i < 10; ++i) {
        P (s1);
        // Critical section
        V (s1);
    } }
```

Semaphore notation

Same as defining a channel and sending a message to it.

```
private static sem s1 = 1;

private static process p1 {
    for (int i = 0; i < 10; ++i) {
        P (s1);
        // Critical section
        V (s1);
    } }
```

Same as `receive s1 ()`

Same as `send s1 ()`

Channels with data

```
private static op void c1 (int);

private static process p1 {
    send c1 (5);
}

private static process p2 {
    int a;
    receive c1 (a);
    System.out.println ("Received message: " + a);
}
```

Channels with data

Each message will contain an `int`

```
private static op void c1 (int);

private static process p1 {
    send c1 (5);
}

private static process p2 {
    int a;
    receive c1 (a);
    System.out.println ("Received message: " + a);
}
```

Channels with data

Each message will contain an `int`

```
private static op void c1 (int);

private static process p1 {
    send c1 (5);
}

private static process p2 {
    int a;
    receive c1 (a);
    System.out.println ("Received message: " + a);
}
```

Sending `5` over the channel

`receive` takes a variable and binds it to the received value

Channels with data (cont.)

```
private static op void c1 (type1, type2, ...);
```

Channels with data (cont.)

Possible to define a channel taking many values. Syntax — like method declaration.

```
private static op void c1 (type1, type2, ...);
```

Channels — queues

```
private static op void c1 (int);

public static void main (String[] args) {
    int a;
    send c1 (3);
    send c1 (4);
    send c1 (2);
    send c1 (7);
    for (int i = 0; i < 4; ++i) {
        receive c1 (a);
        System.out.println ("Received message: " + a);
    }
}
```

Summary

- Semaphores using message passing
- Channels with data
- Using channels as queues

op body

```
private static op void c1 ();

private static void c1 () {
    System.out.println ("Called c1");
}
```

op body

Method with the same name as a channel gets called every time a message is sent to the channel.

```
private static op void c1 ();

private static void c1 () {
    System.out.println ("Called c1");
}
```

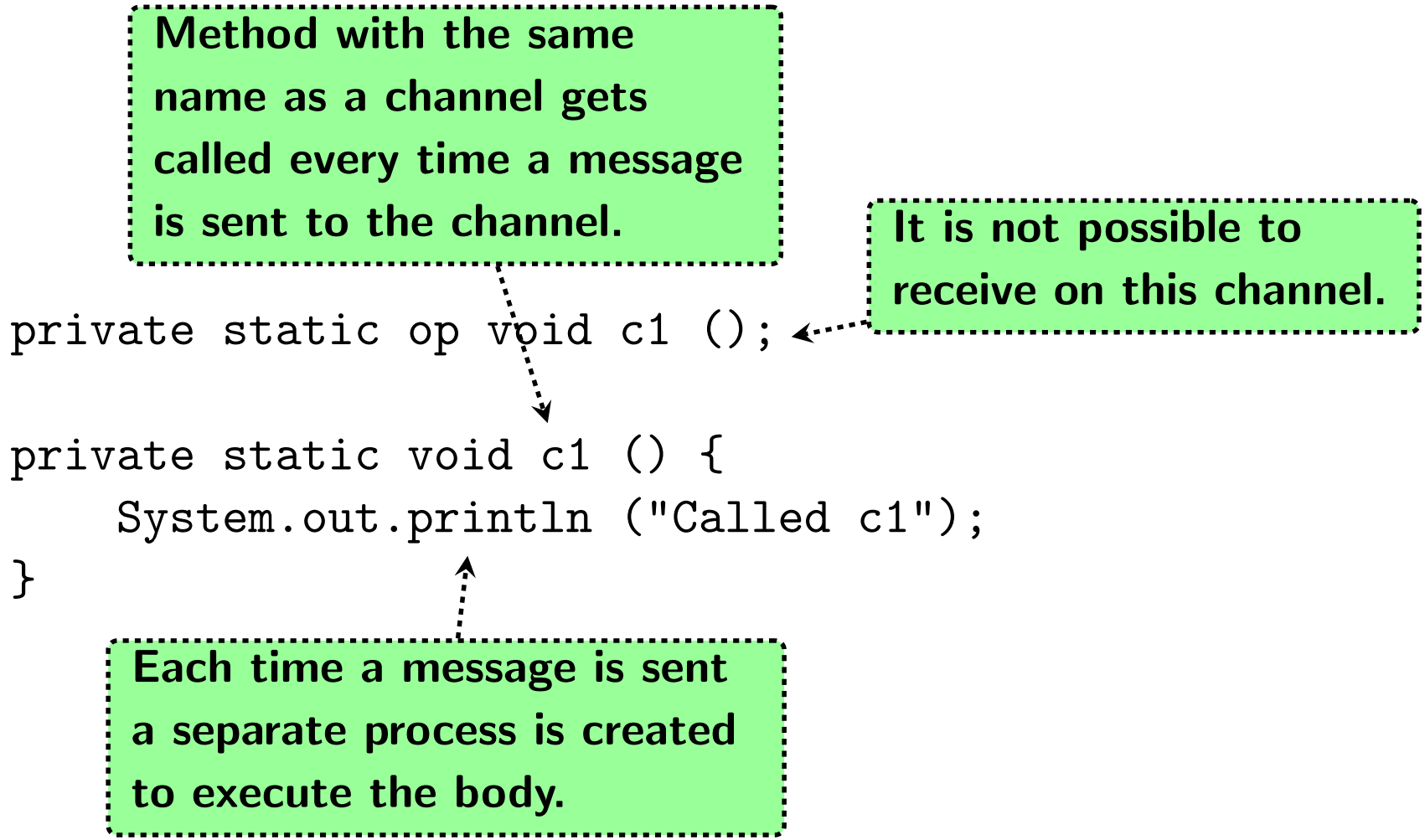
op body

Method with the same name as a channel gets called every time a message is sent to the channel.

```
private static op void c1 ();

private static void c1 () {
    System.out.println ("Called c1");
}
```

Each time a message is sent a separate process is created to execute the body.

op body


Method with the same name as a channel gets called every time a message is sent to the channel.
It is not possible to receive on this channel.
private static op void c1 ();
private static void c1 () {
System.out.println ("Called c1");
}
Each time a message is sent a separate process is created to execute the body.

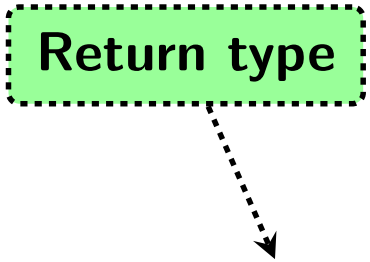
op body (cont.)

- It is possible to write the declaration and definition of an op together.
- `call` on a channel serviced like this will wait until the method finishes.
- op bodies are not so useful (many instances can execute at the same time)

Return type

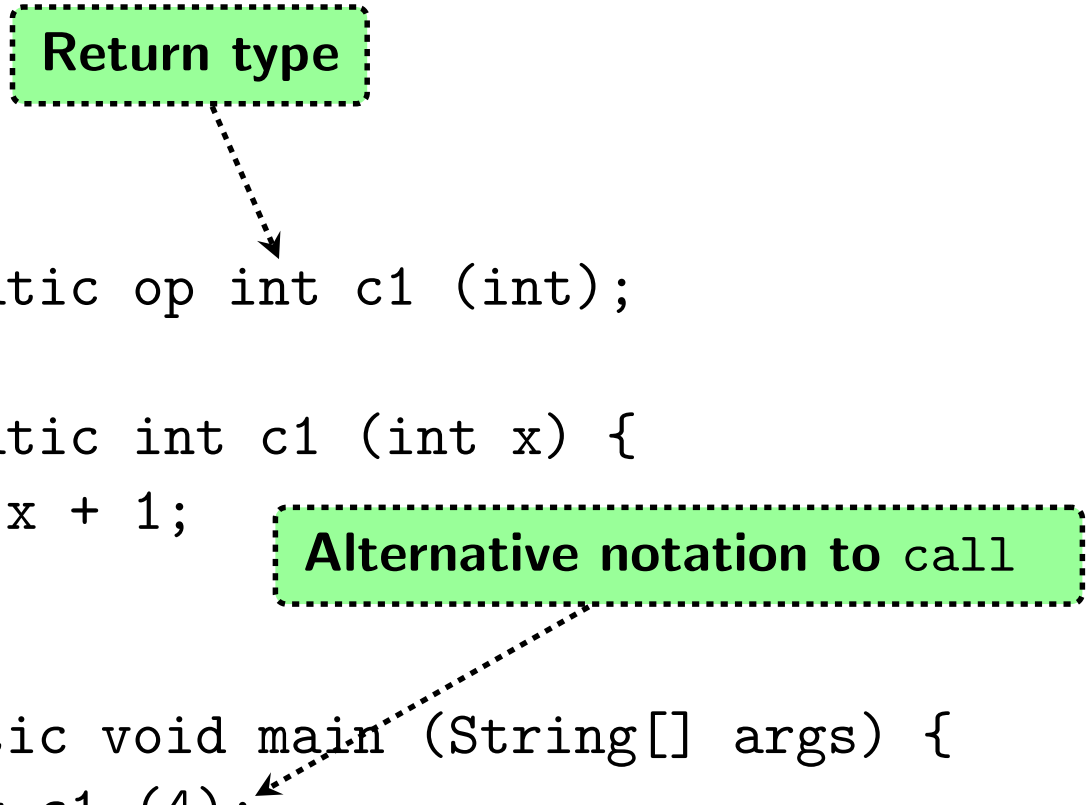
```
private static op int c1 (int);

private static int c1 (int x) {
    return x + 1;
}

public static void main (String[] args) {
    int y = c1 (4);
    System.out.println ("y = " + y);
}
```

Return type




```
private static op int c1 (int);

private static int c1 (int x) {
    return x + 1;
}

public static void main (String[] args) {
    int y = c1 (4);
    System.out.println ("y = " + y);
}
```

Return type


Return type
private static op int c1 (int);
private static int c1 (int x) {
return x + 1;
}
Alternative notation to call
public static void main (String[] args) {
int y = c1 (4);
System.out.println ("y = " + y);
}

Ways of calling

- `send` + `receive`: Asynchronous message
- `call` + `receive`: RDV, no return value
- `call` + `op` body: synchronous call, return value possible

`inni` statement

Is it possible to `receive` and still return a value?

`inni` statement

Is it possible to `receive` and still return a value?
Yes — using `inni`, which is a (very large) extension of `receive`.

inni statement syntax

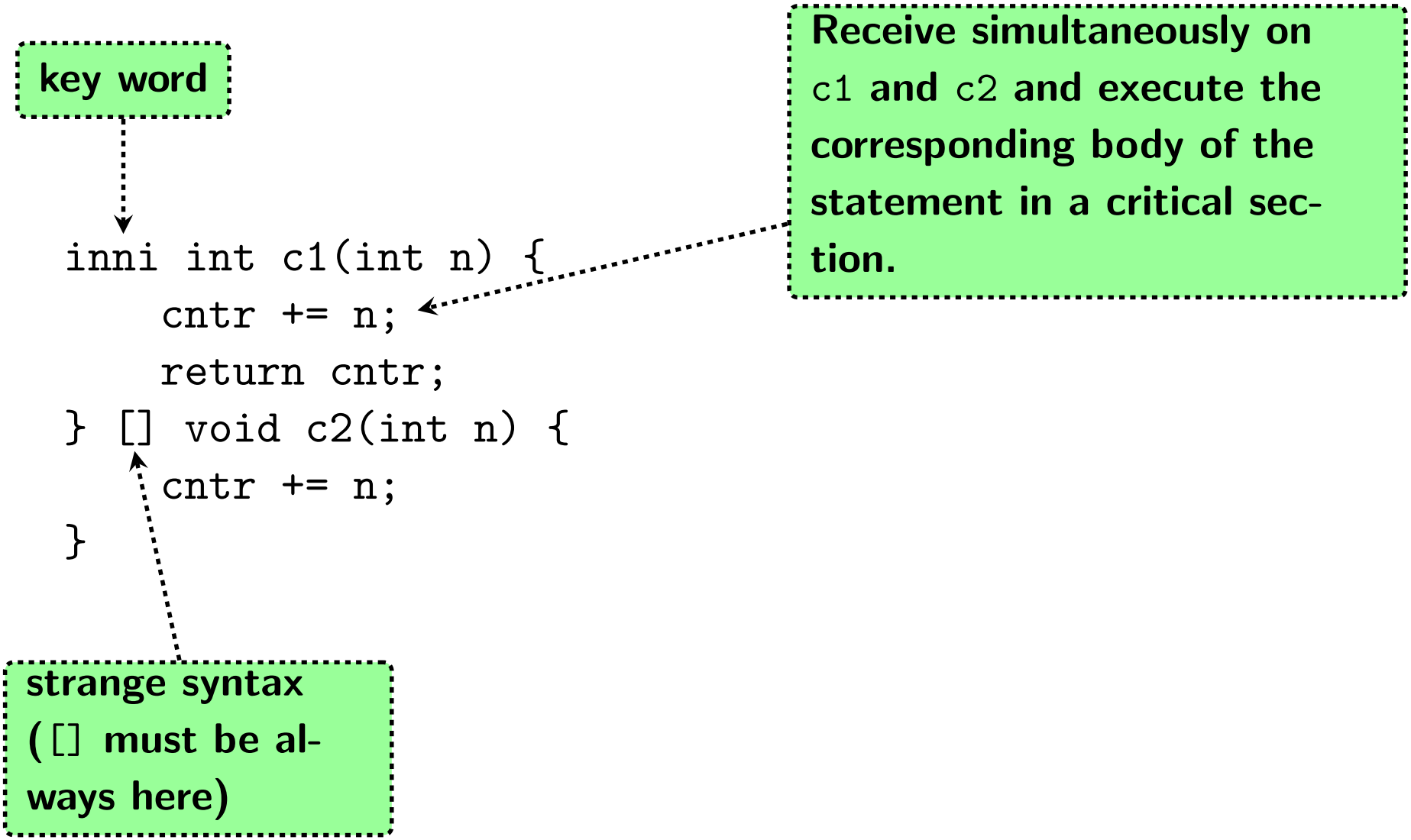
```
inni int c1(int n) {
    cntr += n;
    return cntr;
} [] void c2(int n) {
    cntr += n;
}
```

inni statement syntax

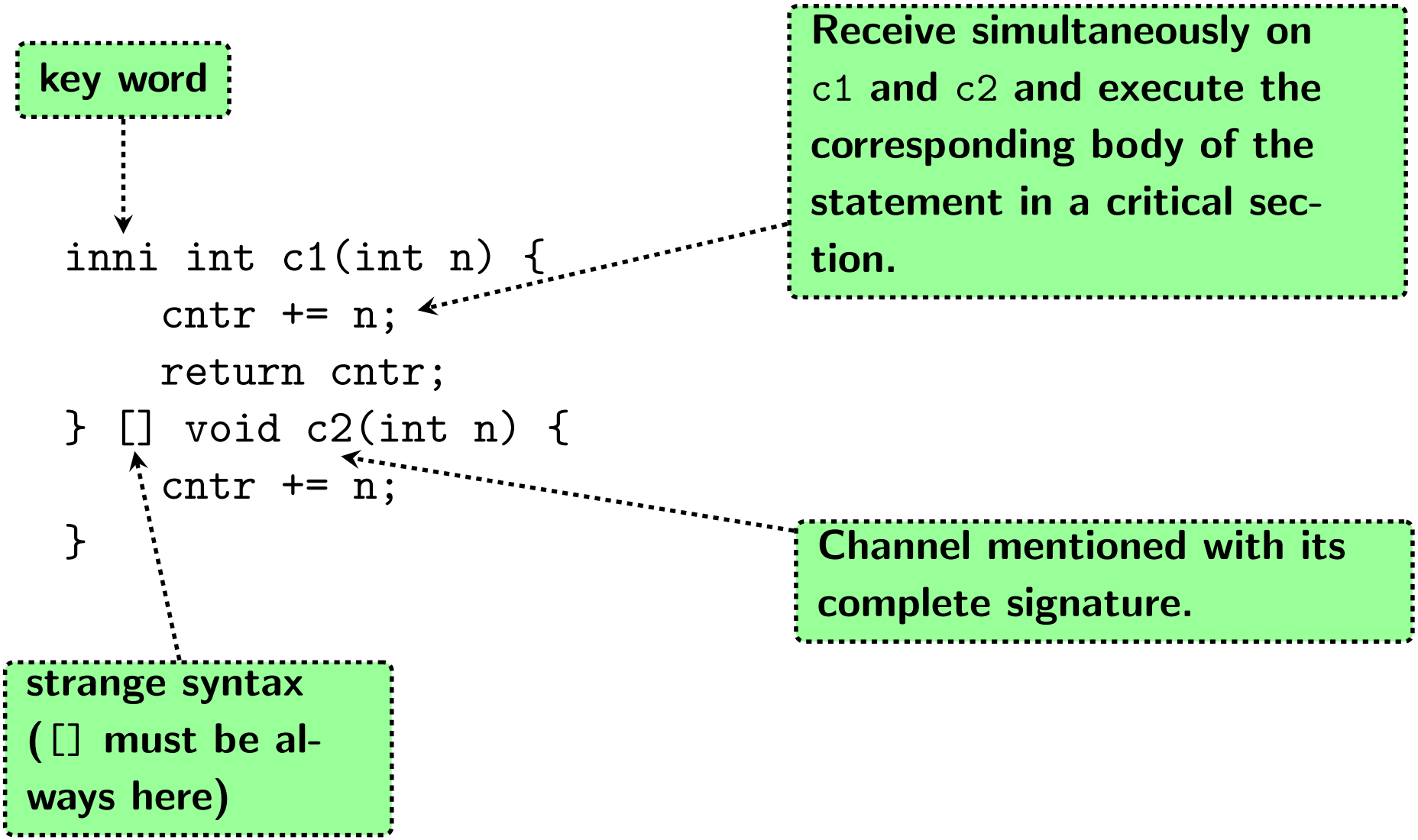
```
inni int c1(int n) {
    cntr += n;
    return cntr;
} [] void c2(int n) {
    cntr += n;
}
```

Receive simultaneously on `c1` and `c2` and execute the corresponding body of the statement in a critical section.

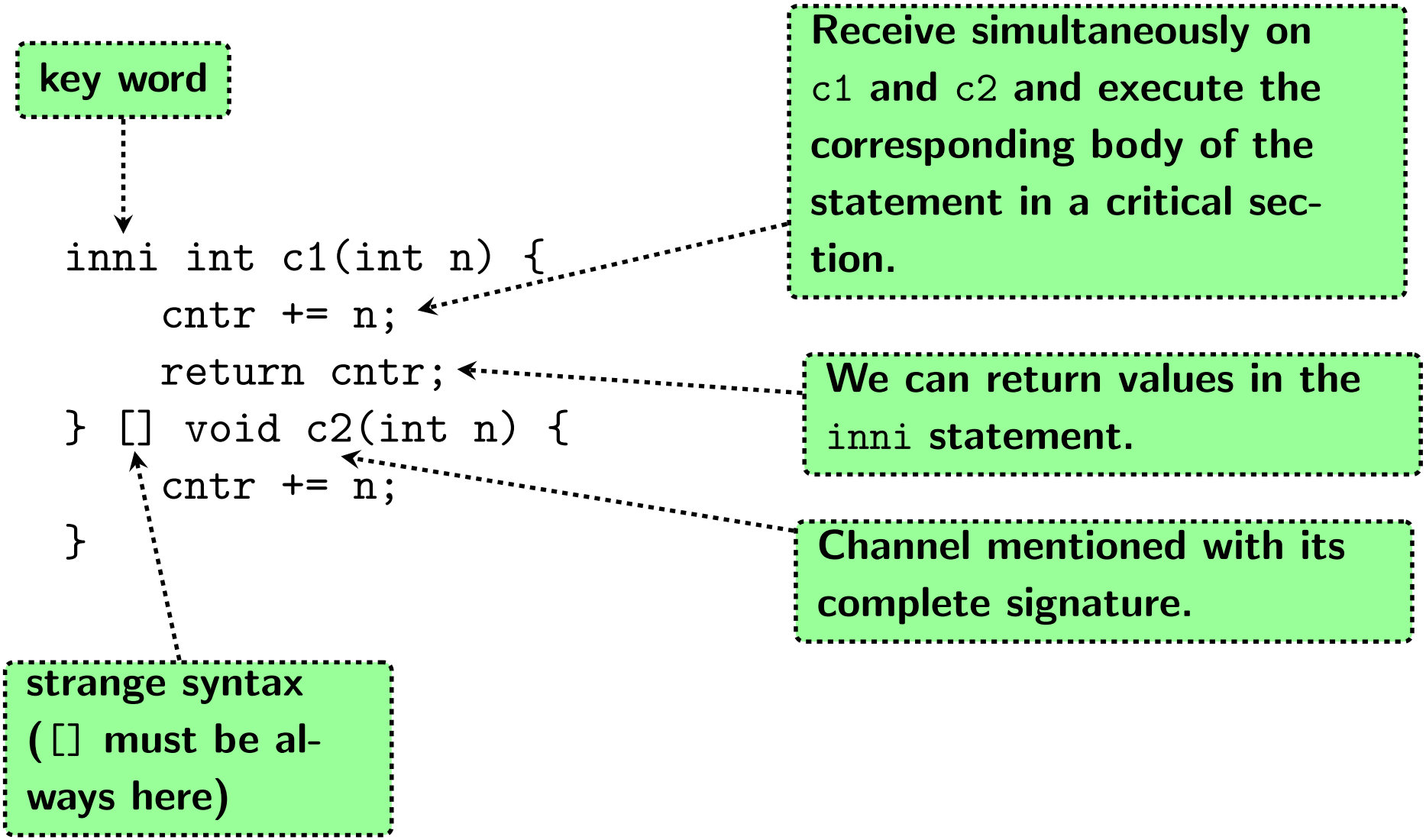
inni statement syntax



inni statement syntax



inni statement syntax



Allocator in Java

```
int allocate (int n) {
  lock.lock ();
  try {
    while (units < n) added.await ();
    return take(n);
  } finally {
    lock.unlock();
  } }
void release (int us) {
  lock.lock ();
  try {
    units += us;
    added.signalAll();
  } finally {
    lock.unlock();
  } }
```

Allocator in Java

```
int allocate (int n) {
  lock.lock ();
  try {
    while (units < n) added.await ();
    return take(n);
  } finally {
    lock.unlock();
  } }
void release (int us) {
  lock.lock ();
  try {
    units += us;
    added.signalAll();
  } finally {
    lock.unlock();
  } }
```

Not enough elements, we have to wait on a condition variable.

Allocator in Java

```
int allocate (int n) {
  lock.lock ();
  try {
    while (units < n) added.await ();
    return take(n);
  } finally {
    lock.unlock();
  } }
void release (int us) {
  lock.lock ();
  try {
    units += us;
    added.signalAll();
  } finally {
    lock.unlock();
  } }
```

Not enough elements, we have to wait on a condition variable.

We need to recheck the condition whenever we wake up.

Allocator in Java

```
int allocate (int n) {
  lock.lock ();
  try {
    while (units < n) added.await ();
    return take(n);
  } finally {
    lock.unlock();
  } }
void release (int us) {
  lock.lock ();
  try {
    units += us;
    added.signalAll();
  } finally {
    lock.unlock();
  } }
```

Not enough elements, we have to wait on a condition variable.

We need to recheck the condition whenever we wake up.

Perhaps somebody is waiting; wake everyone up.

Allocator in JR

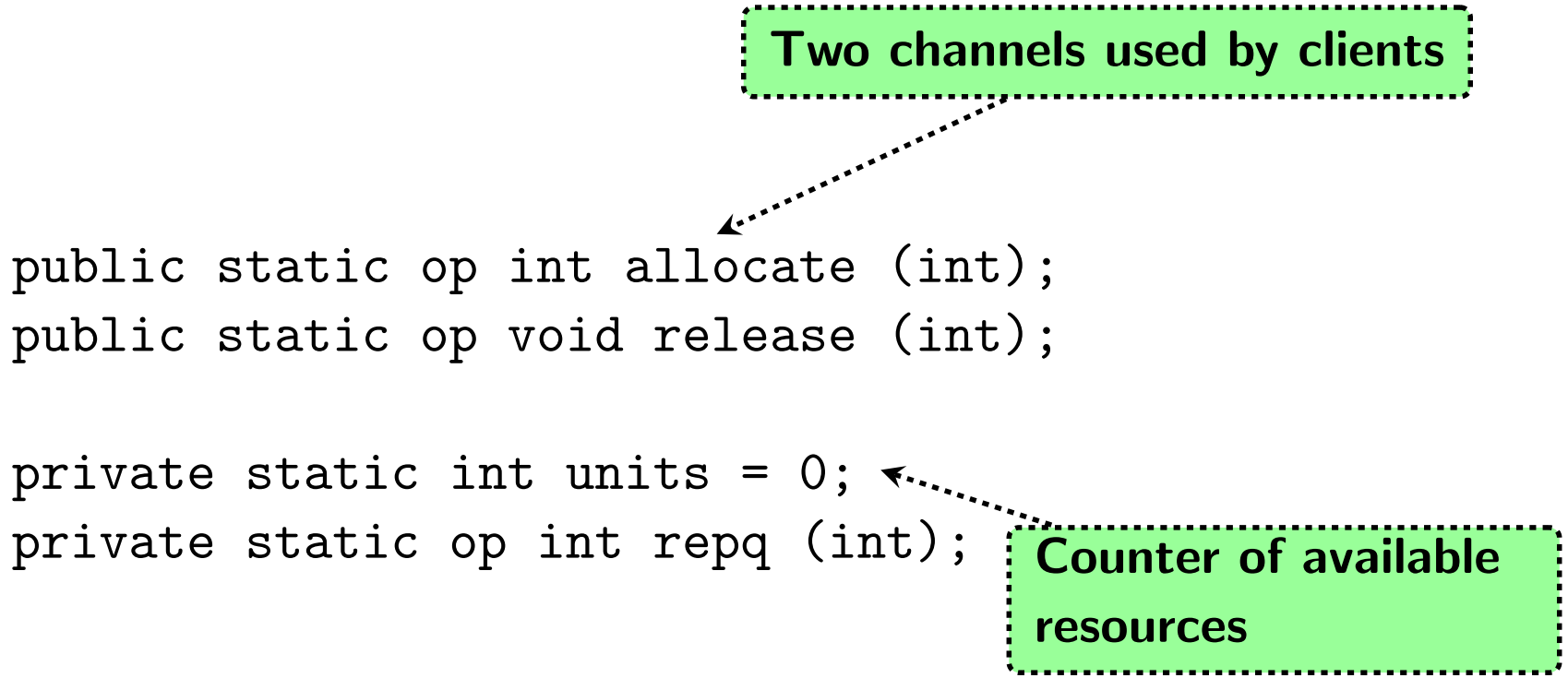
```
public static op int allocate (int);
public static op void release (int);

private static int units = 0;
private static op int repq (int);
```

Allocator in JR

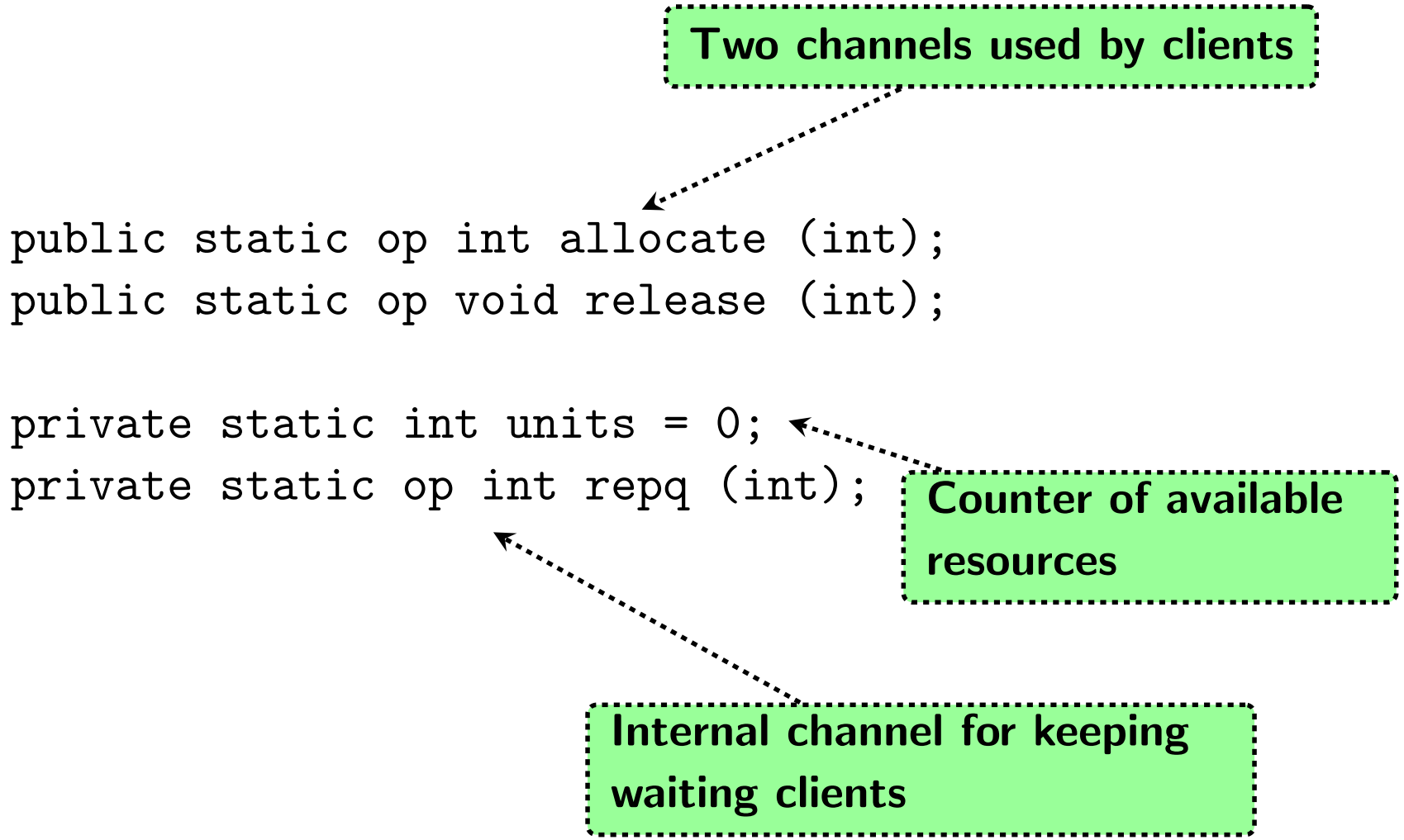
Two channels used by clients

```
public static op int allocate (int);
public static op void release (int);

private static int units = 0;
private static op int repq (int);
```

Allocator in JR



Allocator in JR



Allocator in JR

```
private static process p1 {
    while (true) {
        inni int allocate(int n) {
            if (units < n)
                forward repq(n);
            else
                units -= n;
                return n;
        } [] void release(int us) {
            units += us;
            while (repq.length() > 0)
                inni int repq(int n) {
                    forward allocate(n);
                }
        }
    } }
```

Allocator in JR

If there are not enough elements, push the request to the waiting channel.

```
private static process p1 {
    while (true) {
        inni int allocate(int n) {
            if (units < n)
                forward repq(n);
            else
                units -= n;
                return n;
        } [] void release(int us) {
            units += us;
            while (repq.length() > 0)
                inni int repq(int n) {
                    forward allocate(n);
                }
        }
    } }
```

Allocator in JR

```
private static process p1 {
    while (true) {
        inni int allocate(int n) {
            if (units < n)
                forward repq(n);
            else
                units -= n;
                return n;
        } [] void release(int us) {
            units += us;
            while (repq.length() > 0)
                inni int repq(int n) {
                    forward allocate(n);
                }
        }
    } }
```

If there are not enough elements, push the request to the waiting channel.

Equivalent of `signalAll`.

Allocator in JR

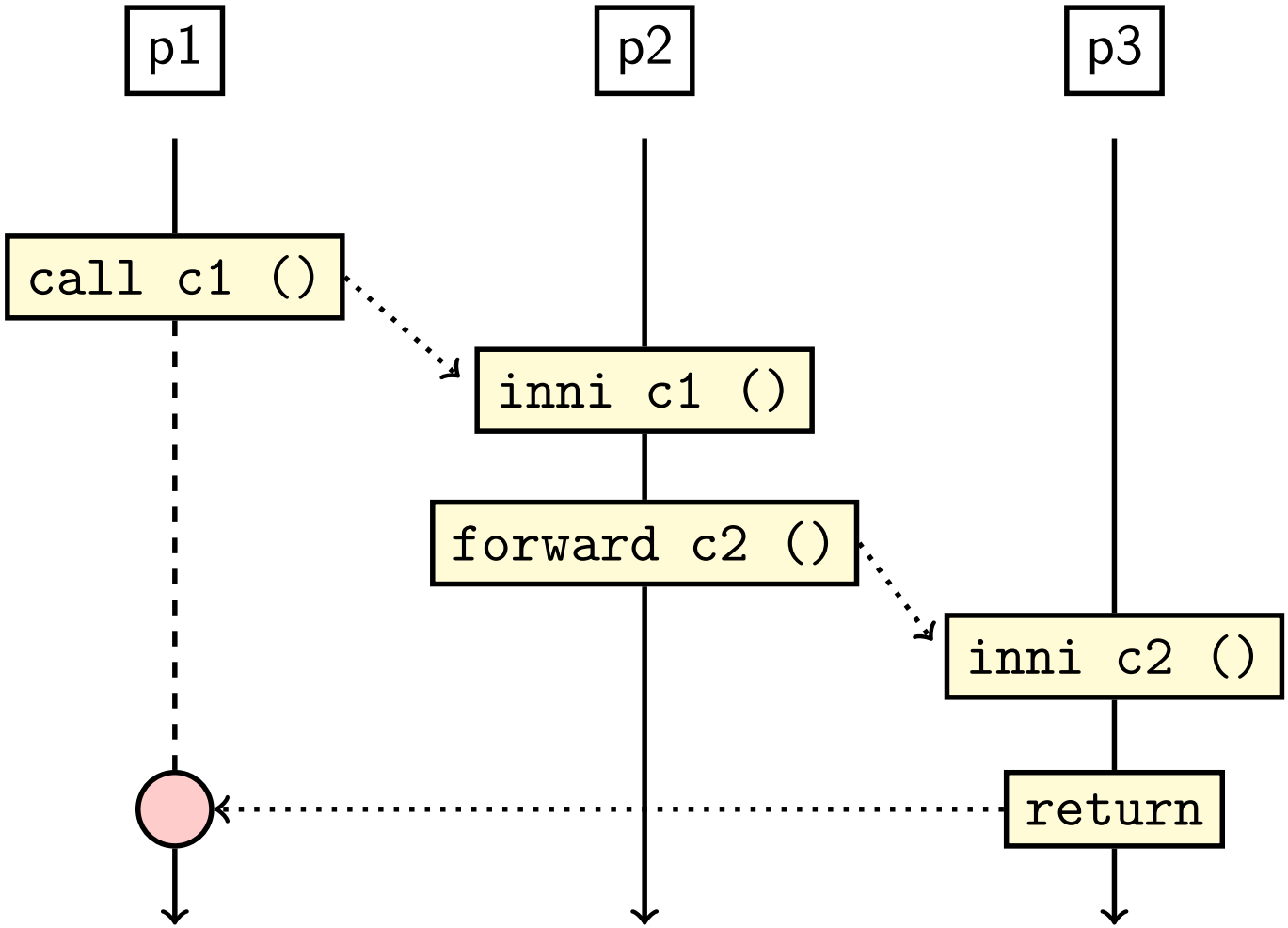
```
private static process p1 {
    while (true) {
        inni int allocate(int n) {
            if (units < n)
                forward repq(n);
            else
                units -= n;
                return n;
        } [] void release(int us) {
            units += us;
            while (repq.length() > 0)
                inni int repq(int n) {
                    forward allocate(n);
                }
        }
    } }
```

If there are not enough elements, push the request to the waiting channel.

Equivalent of `signalAll`.

We jump to the beginning of `allocate` here!.

Forward statement



st clauses

```
inni int allocate(int n) st n <= units {
  units -= n;
  return n;
} [] void release(int n) {
  units += n;
}
```

st clauses

The message will be consumed only if the condition is satisfied.

```
inni int allocate(int n) st n <= units {
  units -= n;
  return n;
} [] void release(int n) {
  units += n;
}
```

st clauses

The message will be consumed only if the condition is satisfied.

```
inni int allocate(int n) st n <= units {
  units -= n;
  return n;
} [] void release(int n) {
  units += n;
}
```

How much simpler it is!

Summary

- Servicing channels with `op` body.
- `inni` statement
- Forwarding messages
- `st` clauses

Message priorities

```
inni int allocate(int n) st n <= units by n {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

Message priorities

Messages with lowest `n` will be handled first.

```
inni int allocate(int n) st n <= units by n {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

Message priorities

Messages with lowest `n` will be handled first.

```
inni int allocate(int n) st n <= units by n {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

Priorities don't work accross branches!

Message priorities (cont.)

```
inni int allocate(int n) st n <= units &&
                    release.length() == 0 {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

Message priorities (cont.)

Receive only if there are no messages in the other channel (useless in this example).

```
inni int allocate(int n) st n <= units &&
                    release.length() == 0 {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

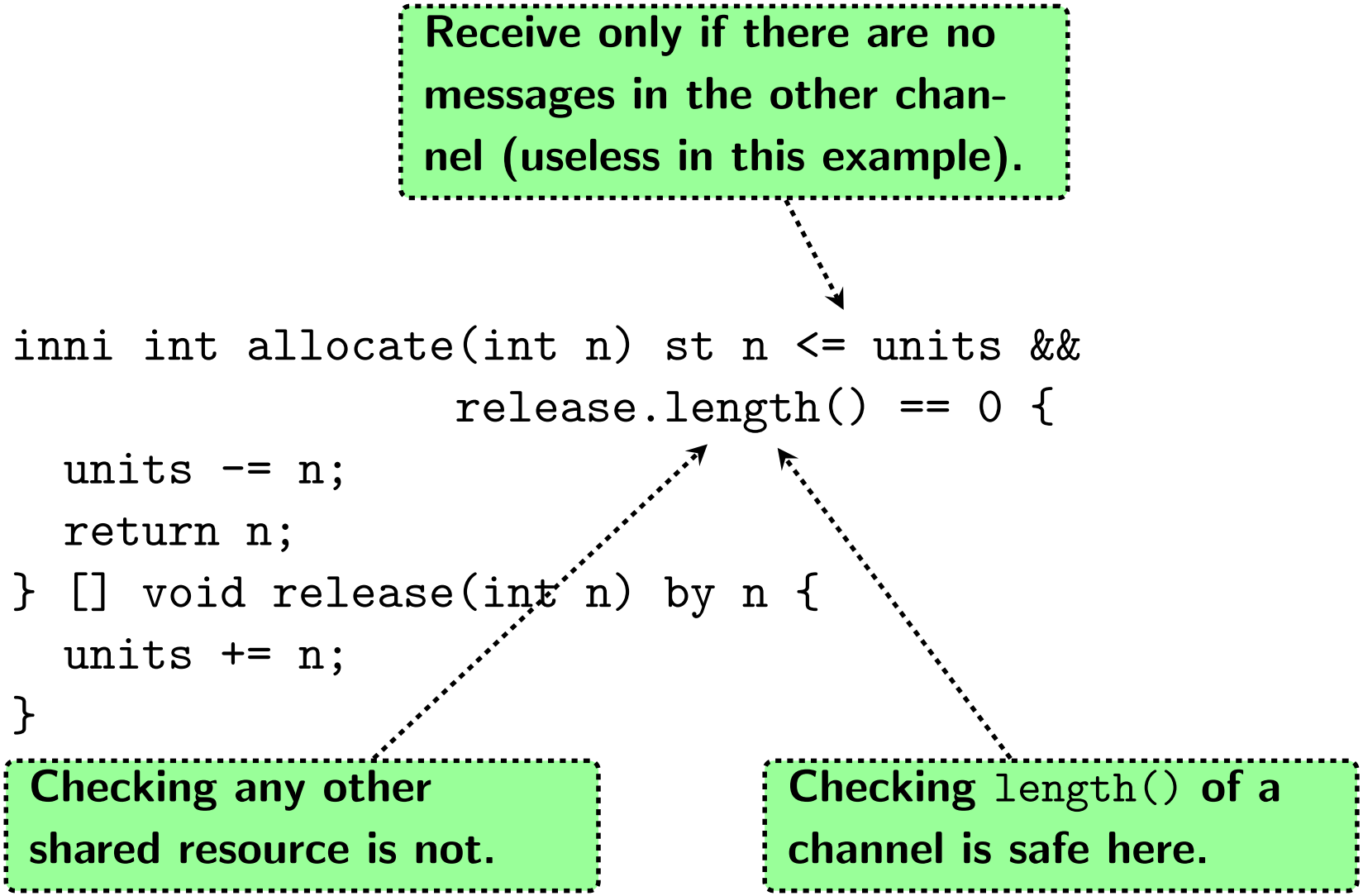
Message priorities (cont.)

Receive only if there are no messages in the other channel (useless in this example).

```
inni int allocate(int n) st n <= units &&
                    release.length() == 0 {
  units -= n;
  return n;
} [] void release(int n) by n {
  units += n;
}
```

Checking `length()` of a channel is safe here.

Message priorities (cont.)



Terminating processes

```
while (run) {
  inni void terminate() {
    run = false;
  } [] else {
  // some work
  }
}
```

Terminating processes

`inni` **statement with an** `else` **branch will check if there is a message in the queue (and receive it).**

```
while (run) {
  inni void terminate() {
    run = false;
  } [] else {
  // some work
  }
}
```

Terminating processes

`inni` **statement with an** `else` **branch will check if there is a message in the queue (and receive it).**

```
while (run) {
  inni void terminate() {
    run = false;
  } [] else {
  // some work
  }
}
```

Terminating processes is a tricky topic.

Terminating processes

`inni` **statement with an** `else` **branch will check if there is a message in the queue (and receive it).**

```
while (run) {
  inni void terminate() {
    run = false;
  } [] else {
  // some work
  }
}
```

You will have to make it work with your program logic.

Terminating processes is a tricky topic.

Capabilities (references to channels)

```
private static op void c1 ();

private static void c1 () {
    cap void () x;
    x = c1;
    receive x ();
}
```

Capabilities (references to channels)

```
private static op void c1 ();

private static void c1 () {
    cap void () x;
    x = c1;
    receive x ();
}
```

Different syntax than `op` declarations (name comes last).

Capabilities (references to channels)

```
private static op void c1 ();

private static void c1 () {
    cap void () x;
    x = c1;
    receive x ();
}

op void c2 (cap void ());
```

Different syntax than `op` declarations (name comes last).

Capabilities (references to channels)

```
private static op void c1 ();

private static void c1 () {
    cap void () x;
    x = c1;
    receive x ();
}


op void c2 (cap void ());
```

Different syntax than `op` declarations (name comes last).

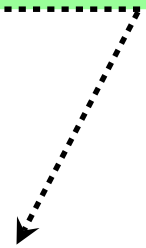
Channel taking a channel reference.

Array of processes

```
private static process p1 ((int i = 0; i < 10; ++i)) {
    // ...
}
```

Array of processes

10 processes doing the same thing.



```
private static process p1 ((int i = 0; i < 10; ++i)) {
    // ...
}
```