

Concurrency, Part 2

by Jørgen Staunstrup, jst@itu.dk, draft version Jan 28, 2021

Resume of part1

This section introduced the stream concept for concurrency programming and exemplified its use for handling three fundamental motivations for using concurrency. In this next section, we will discuss coordination of streams illustrated with code for a simple Stopwatch.

Stream coordination

If streams were completely independent, they would be of limited use. As with people, most of the time they go about their daily business independently of other people. However, every now and then they need to coordinate their activities, e.g. share some information, enter private areas (like a toilet), compete for limited resources etc. Similarly, streams also need to coordinate. This has led to the introduction of a number of programming concepts allowing streams to interact. As with the terminology for concurrency in general, the terminology used for coordination is often confusing and inconsistent.

There are two main types of coordination of streams in software: sharing information and exchanging information. These concepts can be also found in human coordination. An example of sharing information would be a bulletin board. Sending and receiving messages/letters is an example exchanging information.

For computers, there is also a very physical manifestation of the two types of coordination. In some cases, computers may physically share some or all of their memory. In other cases, they may be connected with a network, where information can only be exchanged via some form of communication link.

There have been numerous attempts to introduce programming languages strictly focusing on one of the two types of coordination. For example, Concurrent Pascal was an early object-oriented language where all coordination had to be done through shared objects. Other languages/concepts for concurrency were based on message passing, as the only way for streams to coordinate. An example would be the Http protocol.

Theoretically, sharing objects and message passing are equally powerful. One can always simulate one of them with the other. Therefore, the more general term *coordination* is used here.

Stream coordination

As mentioned above there are two conceptually different ways of coordinating (concurrent) streams:

Shared objects:

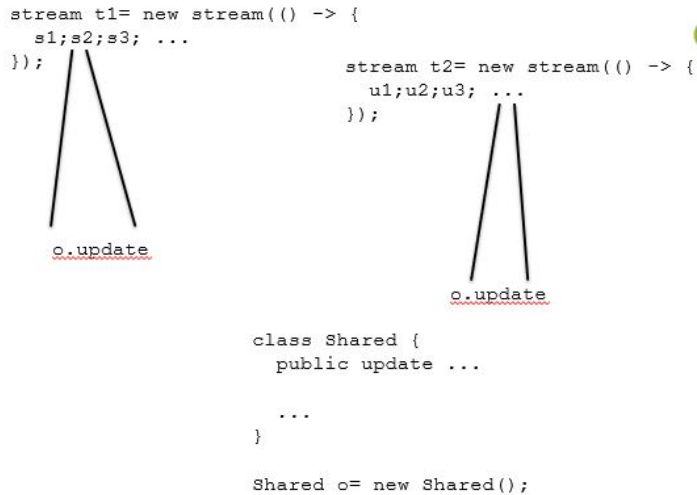
```
stream t1= new stream() -> {
    s1;s2;s3; ...
};

stream t2= new stream() -> {
    u1;u2;u3; ...
};

class Shared {
    public update ...

    ...
}

Shared o= new Shared();
```



Here the object o (of type Shared) can be referenced from both of the streams t1 and t2. They can coordinate their progress by calling methods or referencing public fields in the shared object. This may easily introduce errors e.g. if two streams try to update the same variable in the object simultaneously. We will return to this challenge later.

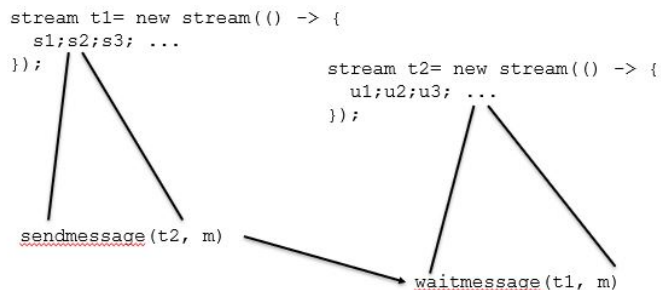
Message passing:

```
stream t1= new stream() -> {
    s1;s2;s3; ...
};

stream t2= new stream() -> {
    u1;u2;u3; ...
};

sendmessage(t2, m)

waitmessage(t1, m)
```



Here the two streams coordinate their progress by sending "messages" to each other. This is a well known (and old) concept. People have been sending letters to each other for centuries, more recent examples are SMS and e-mail. A number of programming languages have built-in mechanisms for message passing:

```
sendmessage(receiver, message);
waitmessage(sender, message);
```

When using message passing (only) between the streams, one avoids the challenges introduced by shared objects such as simultaneous updating. However, as we shall see later, there are other challenges when using message passing.