Consumers

Exercise Notes

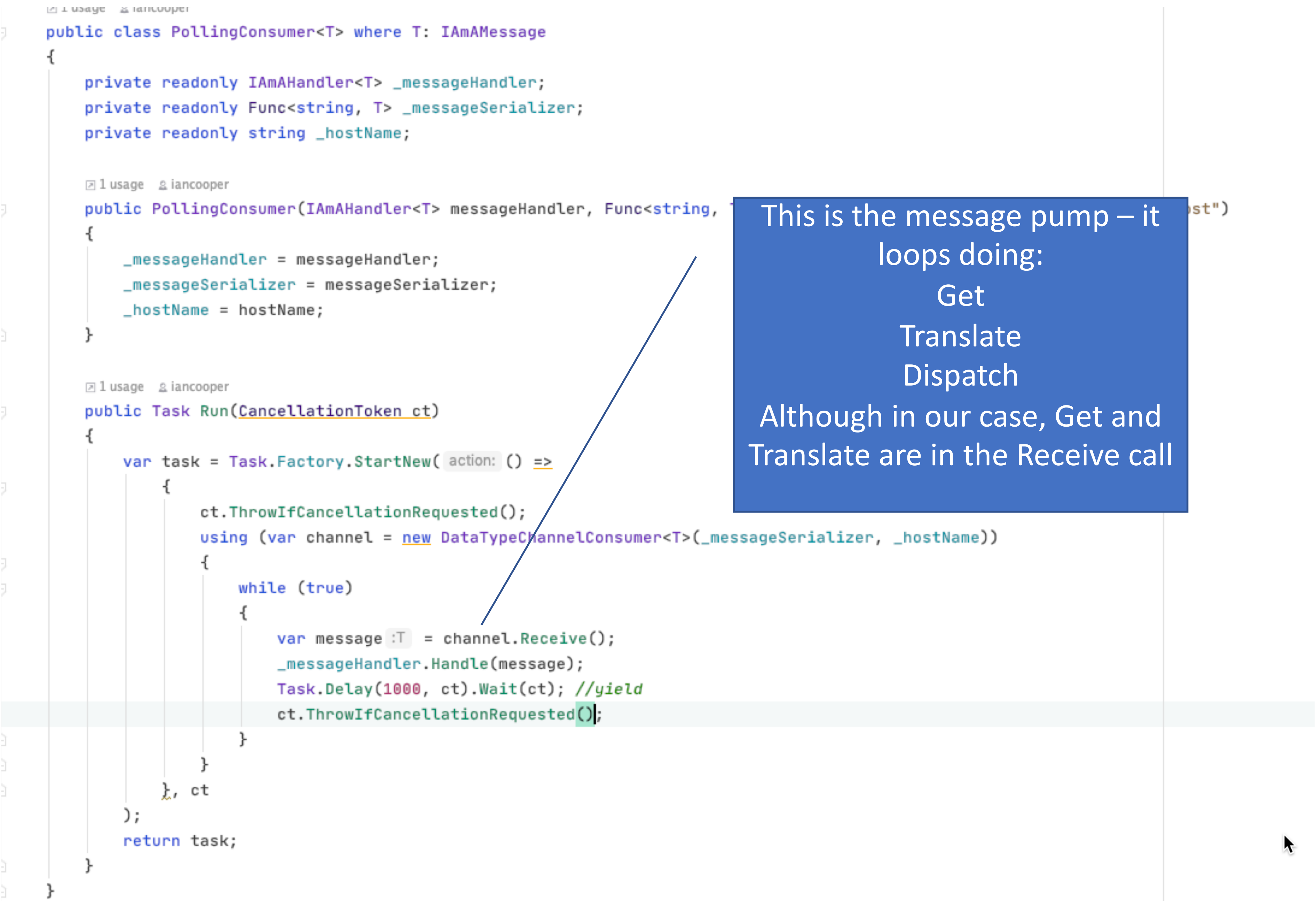
Introduction

- In this exercise you will create a Consumer
- We use a Polling Consumer in C# and Python
  - This is less-efficient on the client than an event-based consumer
  - Though it is less burden on the server
    - Middleware like SNS/SQS on AWS does not offer an event consumer option, it’s all polling
- We use an Event Consumer in JavaScript
  - This can be a more complex programming model
  - Applying ‘backpressure’ – slowing down the rate of arrival of messages to protect downstream resources – can be more complex or not possible depending on the middleware API
  - RMQ lets us use QoS to change this

Notes

- Up to now we have sent and received messages on demand
  - A real world application tends to wait for messages to arrive
  - We want to separate user code – a handler – from the endpoint code
- We create a Message Pump to read messages
    - This implementation has the relevant steps though is cruder than a real pump
    - As we only have one translator and handler, we can just use them without lookup
- A handler and translator is usually what the application developer writes
  - The endpoint code is the pump and gateway over the middleware API

C#



```
1 usage   iancooper
public class PollingConsumer<T> where T: IAmAMessage
{
    private readonly IAmAHandler<T> _messageHandler;
    private readonly Func<string, T> _messageSerializer;
    private readonly string _hostName;

    1 usage   iancooper
    public PollingConsumer(IAmAHandler<T> messageHandler, Func<string,                                  st")
    {
        _messageHandler = messageHandler;
        _messageSerializer = messageSerializer;
        _hostName = hostName;
    }

    1 usage   iancooper
    public Task Run(CancellationToken ct)
    {
        var task = Task.Factory.StartNew( action: () =>
            {
                ct.ThrowIfCancellationRequested();
                using (var channel = new DataTypeChannelConsumer<T>(_messageSerializer, _hostName))
                {
                    while (true)
                    {
                        var message :T  = channel.Receive();
                        _messageHandler.Handle(message);
                        Task.Delay(1000, ct).Wait(ct); //yield
                        ct.ThrowIfCancellationRequested();
                    }
                }
            }, ct
        );
        return task;
    }
}
```

This is the message pump – it loops doing:

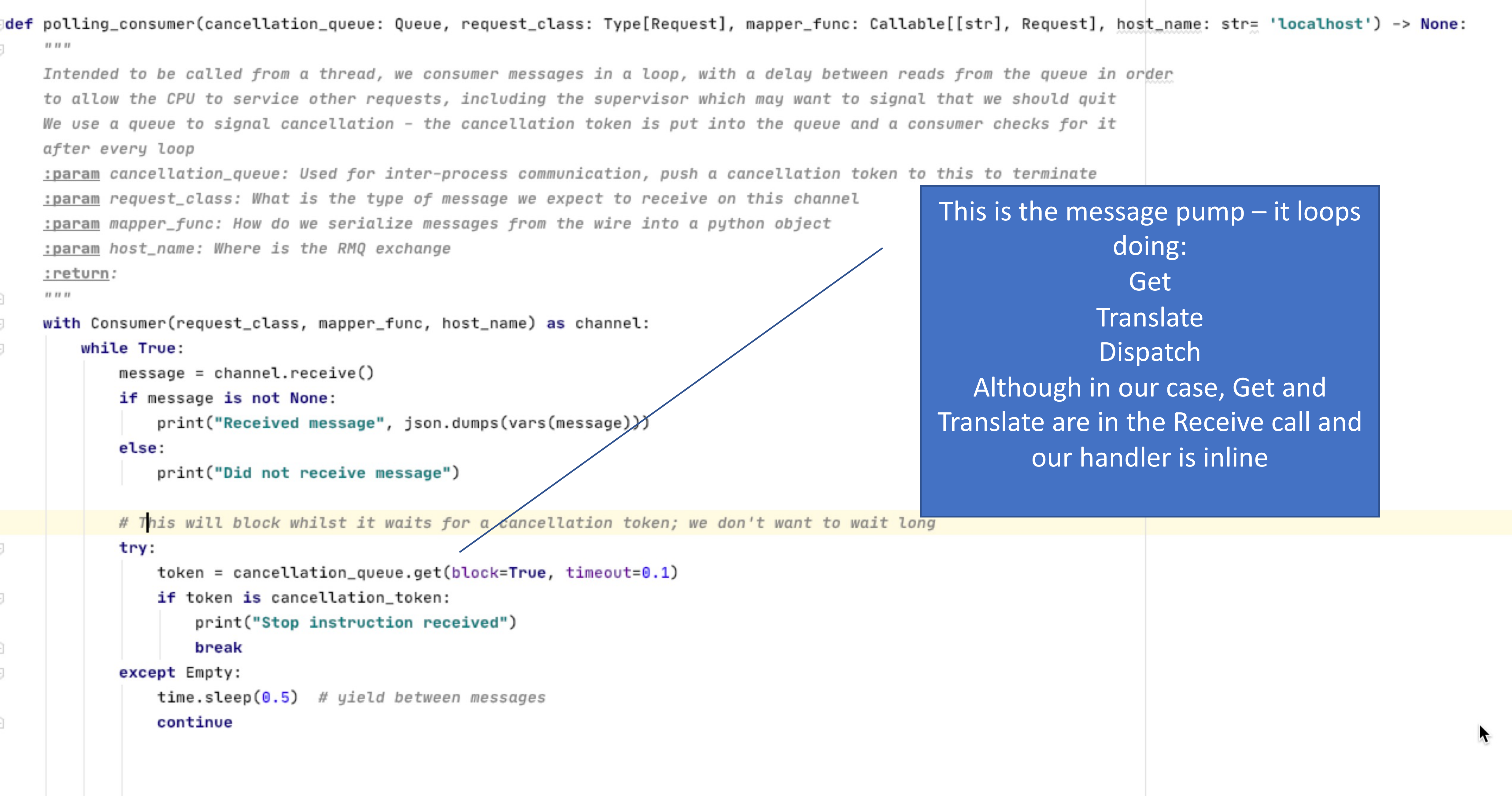
Get

Translate

Dispatch

Although in our case, Get and Translate are in the Receive call

Python

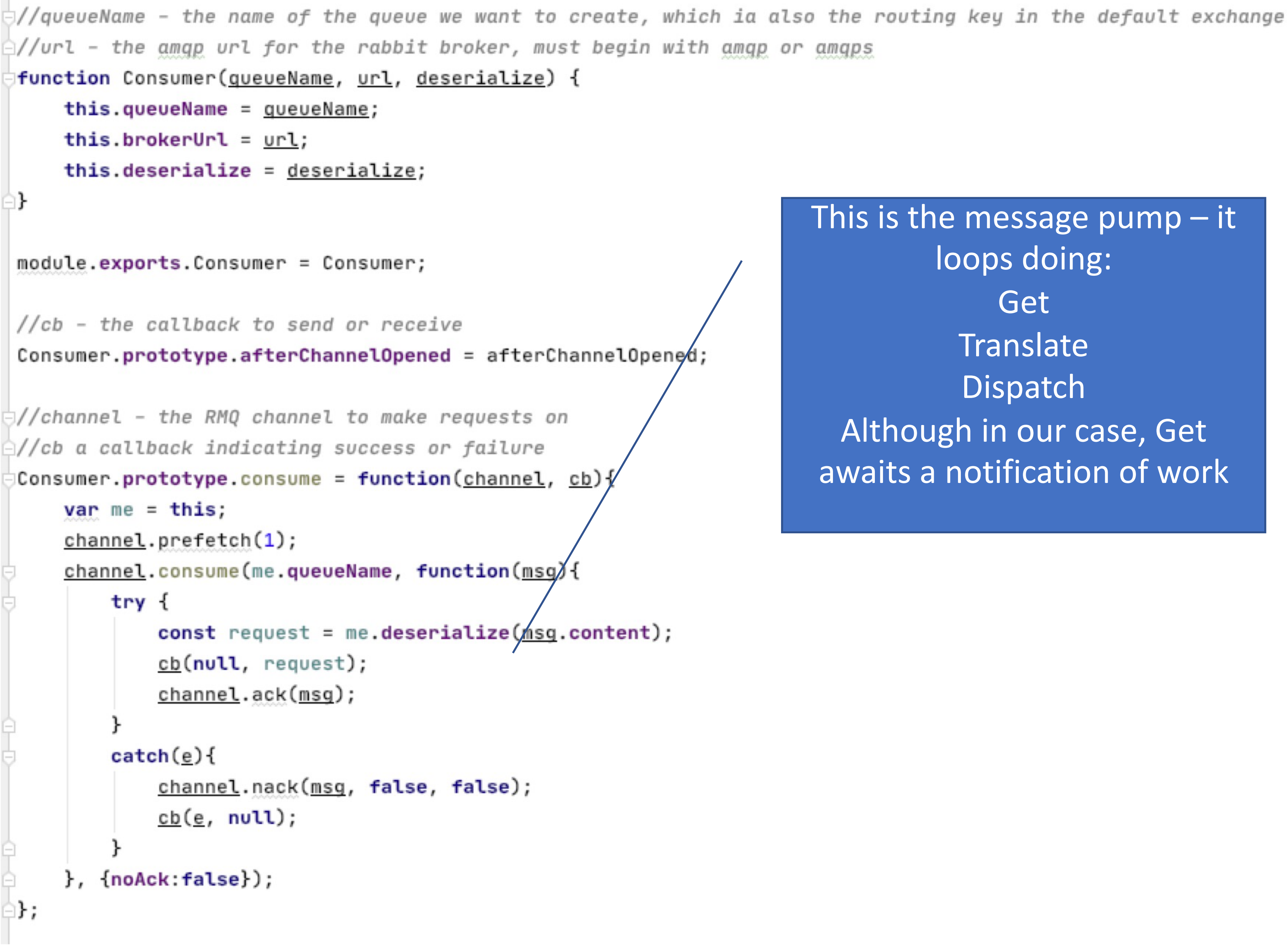


```python
def polling_consumer(cancellation_queue: Queue, request_class: Type[Request], mapper_func: Callable[[str], Request], host_name: str= 'localhost') -> None:
    """
    Intended to be called from a thread, we consumer messages in a loop, with a delay between reads from the queue in order
    to allow the CPU to service other requests, including the supervisor which may want to signal that we should quit
    We use a queue to signal cancellation - the cancellation token is put into the queue and a consumer checks for it
    after every loop
    :param cancellation_queue: Used for inter-process communication, push a cancellation token to this to terminate
    :param request_class: What is the type of message we expect to receive on this channel
    :param mapper_func: How do we serialize messages from the wire into a python object
    :param host_name: Where is the RMQ exchange
    :return:
    """
    with Consumer(request_class, mapper_func, host_name) as channel:
        while True:
            message = channel.receive()
            if message is not None:
                print("Received message", json.dumps(vars(message)))
            else:
                print("Did not receive message")

            # This will block whilst it waits for a cancellation token; we don't want to wait long
            try:
                token = cancellation_queue.get(block=True, timeout=0.1)
                if token is cancellation_token:
                    print("Stop instruction received")
                    break
            except Empty:
                time.sleep(0.5)  # yield between messages
                continue
```

This is the message pump – it loops doing:
Get
Translate
Dispatch
Although in our case, Get and Translate are in the Receive call and our handler is inline

JavaScript



```
//queueName - the name of the queue we want to create, which ia also the routing key in the default exchange
//url - the amqp url for the rabbit broker, must begin with amqp or amqps
function Consumer(queueName, url, deserialize) {
    this.queueName = queueName;
    this.brokerUrl = url;
    this.deserialize = deserialize;
}

module.exports.Consumer = Consumer;

//cb - the callback to send or receive
Consumer.prototype.afterChannelOpened = afterChannelOpened;

//channel - the RMQ channel to make requests on
//cb a callback indicating success or failure
Consumer.prototype.consume = function(channel, cb){
    var me = this;
    channel.prefetch(1);
    channel.consume(me.queueName, function(msg){
        try {
            const request = me.deserialize(msg.content);
            cb(null, request);
            channel.ack(msg);
        }
        catch(e){
            channel.nack(msg, false, false);
            cb(e, null);
        }
    }, {noAck:false});
};
```

This is the message pump – it loops doing:

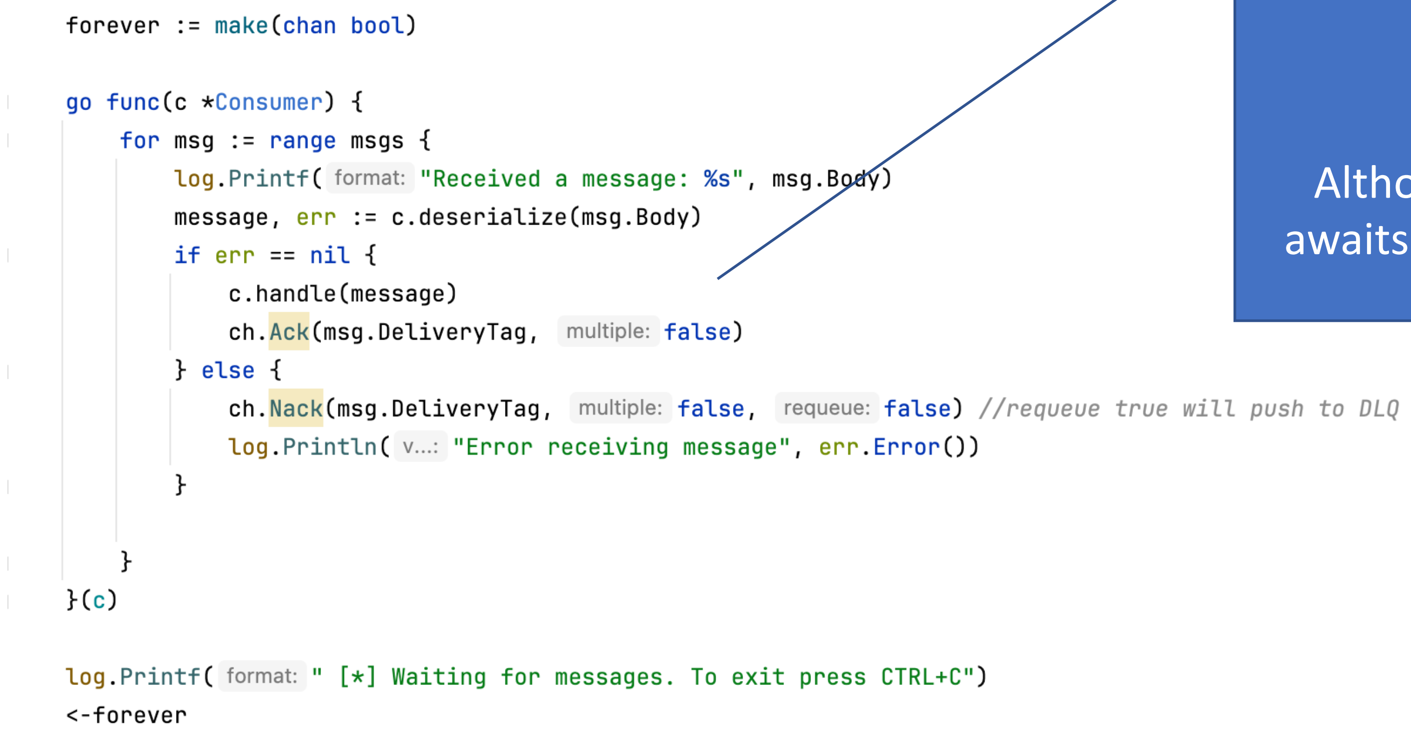
Get

Translate

Dispatch

Although in our case, Get awaits a notification of work

Go



```go
forever := make(chan bool)

go func(c *Consumer) {
    for msg := range msgs {
        log.Printf( format: "Received a message: %s", msg.Body)
        message, err := c.deserialize(msg.Body)
        if err == nil {
            c.handle(message)
            ch.Ack(msg.DeliveryTag,  multiple: false)
        } else {
            ch.Nack(msg.DeliveryTag,  multiple: false,  requeue: false) //requeue true will push to DLQ
            log.Println( v...: "Error receiving message", err.Error())
        }

    }
}(c)

log.Printf( format: " [*] Waiting for messages. To exit press CTRL+C")
<-forever
```

This is the message pump – it loops doing:

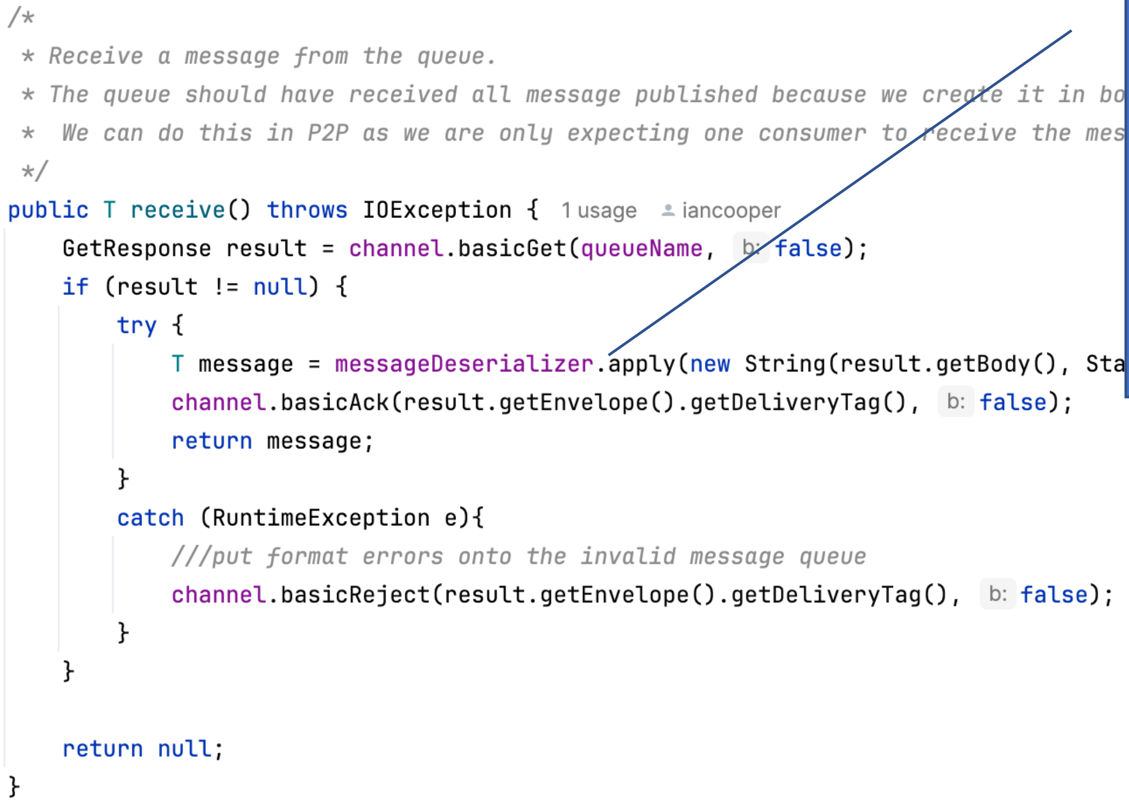
Get

Translate

Dispatch

Although in our case, Get awaits a notification of work

Java



```
/*
 * Receive a message from the queue.
 * The queue should have received all message published because we create it in bo
 *  We can do this in P2P as we are only expecting one consumer to receive the mes
 */
public T receive() throws IOException {  1 usage  iancooper
    GetResponse result = channel.basicGet(queueName,  b: false);
    if (result != null) {
        try {
            T message = messageDeserializer.apply(new String(result.getBody(), Sta
            channel.basicAck(result.getEnvelope().getDeliveryTag(),  b: false);
            return message;
        }
        catch (RuntimeException e){
            ///put format errors onto the invalid message queue
            channel.basicReject(result.getEnvelope().getDeliveryTag(),  b: false);
        }
    }

    return null;
}
```

This is the message pump – it loops doing:
Get
Translate
Dispatch
Although in our case, Get awaits a notification of work