Explain what is meant by the stream abstraction.
Stream abstraction is essentially a way to handle asynchronous events from multiple sources without needing to know exact size, data type or arrival time of the events. A stream handles events at a high, abstracted level, taking in data from potentially multiple sources, of different types, of infinite amounts and allows for the data to be collected and later handled by separate functionality within the program.

What is the relationship between streams and the observer pattern?
Streams extend the observer pattern, so that instead of just performing an action based on a single event, you can create a stream to handle multiple types of events and subscribe to the stream, performing multiple different actions based on the events that are observed in the stream.

What are streams useful for modelling and when might you use them in Rich Web development?
Streams are useful for modelling asynchronous events, where a program flow does not need to wait for a specific event to occur but can merely continue executing and listen for a specific event to occur, then perform some action based on that event. For instance, a stream is useful when detecting user input such as clicks or keypresses from a GUI, where the event arrival time is unknown and at the discretion of the user but is necessary to change some state within the program based on the specific event.

Describe in detail how you could use the RxJS library to handle asynchronous network responses to API requests.
First you need to import a number of functions from RxJS library such as `from`, `pipe`, `map`, `subscribe` and `pluck`. Then create an `Observable` using the `from` method of RxJS, passing in a get request and the URL of the API being used as the source. Include some error handling for the response from the request to ensure it is received successfully before trying to parse the data i.e. HTTP/1.1 Status Code 200 (OK). The RxJS `from` method will return an `Observable` from the request that we can then `subscribe` to and wait for a response to arrive. In the `subscribe` method we can include code to parse the data and call other functions to perform actions based on the data that is received. For instance, you could use `map`, `pipe` and `pluck` to filter out specific pieces of the data returned and use those values to update the `DOM` elements in the app interface or `mapTo` specific function calls.

In your opinion, what are the benefits to using a streams library for networking over, say, promises? Both promises and streams can be used to handle asynchronous tasks and they both handle them in a similar manner. The significant difference between them is that streams provide for the ability to handle multiple asynchronous tasks. Once a promise has resolved its asynchronous task it completes and can no longer be used. For an application that is handling network tasks this is not ideal, for example if you are trying to receive continuous updates of some value in your application, like in the case of connecting to a web socket application with push notifications. In this case, the application requires something more complex than the 'simple' asynchronicity provided by promises and streams solves this problem.

And what do you think are the downsides?
Having only used streams for a number of weeks I cannot speak to their disadvantages with great detail, but I would imagine it is possible that the abstraction provided by stream libraries and the ability to merge asynchronous tasks into one observable object could make a project more difficult to debug and separate out areas of functionality. As streams handle all the tasks it is possible it would be very easy to find yourself using them all over your code and potentially you could find it harder to unpick the functionality at a later date if needed.