Programmazione Avanzata e Paradigmi
Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module-4.2]
ASYNCHRONOUS PROGRAMMING
SUMMARY

• Asynchronous programming
  – based on future/task
  – based on CPS and callbacks
• Event-loop model
• callback hell
  – asynchronous spaghetti, callback nesting
• Promises
  – chaining, composition
• From events to event streams: Reactive programming
  – flapjax
• Reactive extensions (Rx)
  – Observable, observers
REFERENCES

• AsyncJavascript. Trevor Burnham. The Pragmatic Programmers Series


• MSDN - Asynchronous programming on .NET 4.5


• Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In Proceedings of the 7th Workshop on Dynamic Languages and Applications (DYLA '13). ACM, New York, NY, USA


ASYNCHRONOUS PROGRAMMING

• A style of programming getting more and more important today
  – generally speaking, it’s about about the execution and management of asynchronous computations/request/processes
    • abstracting from threads…
  – recently supported by all major programming frameworks and mainstream platforms
    • Java, .NET, iOS, Android..
  – including a variety of mechanisms and architectures
    • including event-driven programming, but not limited to
• Still an evolving issue
  – e.g. Microsoft case: first proposing Asynchronous Programming Model (APM), deprecated then for the Event-based Asynchronous Pattern (EAP) with .NET 2.0, deprecated for the Task-based Asynchronous Pattern (TAP) with .NET 4.5...
ASYNCHRONOUS PROGRAMMING: APPROACHES

• Two main kinds
  – task/future based
  – continuation passing style & event-driven
TASK & FUTURE BASED ASYNC

• **Future** mechanism
  – by issuing an asynchronous computation (task), an object representing the future result or the state of the computation itself is created and immediately returned
  – the object allows for
    • checking the state of the task (poll)
    • blocking when the task result is needed
    • cancel the ongoing task, when possible
    • catching errors/exception related to the ongoing task

• Examples:
  – Java Task Executor framework
  – Microsoft Task-based Asynchronous Pattern
  – Grand-Central Dispatch in iOS
  – AsyncTask in Android
EXAMPLE: JAVA EXECUTORS

- Callable tasks in executors
  - tasks representing a deferred computation, completing with some kind of result
    ```java
    public interface Callable<V> {
        V call() throws Exception;
    }
    ```
  - call method encapsulates the behavior of the task doing some kind of computation and returning some kind \( V \) of value

- Submitted to an executor service by means of submit method, returning a future
  ```java
  public interface ExecutorService extends Executor {
      ...
      void execute(Runnable task);
      Future<T> submit(Callable<T> task);
      ...
  }
  ```
  - future provide methods to test whether the task has completed or been cancelled, to retrieve results (blocking) or cancel the task
EXAMPLE: MICROSOFT TAP
TASK-BASED ASYNCHRONOUS PATTERN

• Microsoft Task-based Asynchronous Pattern (TAP)
  – based on the Task and Task<TResult> types in the System.Threading.Tasks namespace
    • used to represent arbitrary asynchronous operations.
• TAP uses a single method to represent the initiation and completion of an asynchronous operation
  – method returns either a Task or a Task<TResult>, based on whether the corresponding synchronous method returns void or a type TResult.
• Implementing the Task-based Asynchronous Pattern - three ways:
  – by using the C# and Visual Basic compilers in Visual Studio,
  – manually, or
  – through a combination of the compiler and manual methods.
public static Task<int> ReadTask(this Stream stream, byte[] buffer, int offset, int count, object state)
{
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, ar =>
    {
        try { tcs.SetResult(stream.EndRead(ar)); } 
        catch (Exception exc) { tcs.SetException(exc); }
    }, state);
    return tcs.Task;
}
TAP EXAMPLE - WITH ASYNC AND AWAIT

• Language support: async methods
  – methods that can be suspended (by using Await or await operator) releasing the control to the caller
• await operator
  • is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes.
  – the marked async method can itself be awaited by methods that call it

```csharp
private async Task SumPageSizesAsync()
{
    HttpClient client = new HttpClient();
    // . . .
    Task<byte[]> getContentsTask = client.GetBytesArrayAsync(url);
    byte[] urlContents = await getContentsTask;

    // Equivalently, now that you see how it works,
    // you can write the same thing in a single line.
    // byte[] urlContents = await client.GetBytesArrayAsync(url);
    // . . .
}
```
ASYNC PROGRAMMING BASED ON CONTINUATION PASSING STYLE

- Rethinking to the computation & programming model to make it fully based on asynchronous programming
- When triggering the execution of an asynchronous computation, we specify the functions that must be called as "continuation" of the computation when the asynchronous computation has been completed or an error occurred
  - the continuation has one parameter, which is the return value passed by the asynchronous computation

```javascript
// pseudo-definition
function myAsyncFunc(param1, ..., paramN, continuation) {
    ...
    // when the result of myAsyncFunc is ready, then
    // continuation(result) is called
}

// non blocking invocation
myAsync(..., cont_function(value) {...})
```
CONTINUATION PASSING STYLE (CPS) IN GENERAL

• In functional programming, continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation.
  – first introduced by Gerald Jay Sussman and Guy L. Steele, Jr. 1975 working on Scheme

• A function written in continuation-passing style takes an extra argument, a function of one argument.
  – i.e. the continuation

• When the CPS function has computed its result value, it "returns" it by calling the continuation function with this value as the argument
  – that means that when invoking a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value.
CALLBACKS AS CONTINUATIONS

- In asynchronous programming, callbacks are continuations that are called when the result of the asynchronous computation is ready
  - examples:
    - the response of an asynchronous request to a remote service has arrived
    - the result of a I/O task is available
    - ...
- Abstract example in JavaScript:

```javascript
// synch version

function loadUserPic(userId)
    { var user = findUserById(userId);
      return loadPic(user.picId);
    }

// async version -
// both findUserById & loadUserPic
// are async processes */

function loadUserPic(userId, ret)
    { findUserById(userId, function(user) {
        loadPic(user.picId, ret);
      });
    }

loadUserPic('john', function(pic)
    { ui.show(pic);
      });
```
EXECUTION MODEL

• A main question is:

  *who invokes the continuation/callback?*

• Two main possibilities:
  1. a separate thread of control, running concurrently to the thread of control that triggered the request, possibly concurrently
     • inversion of control
     • problems: races
  2. the same logical thread of control that triggered the request:
     **event loop model**
     • execution model of modern web apps (client/server) and adopted in event-driven programming in general
EVENT-LOOP CONTROL ARCHITECTURE AND EVENT-DRIVEN PROGRAMMING

• Behaviour organized as a set of event handlers
  – encapsulating the computation to execute when some event is perceived
• An event queue is used to keep track of events generated by either the environment or the event handler themselves
• Program behaviour conceptually represented by an loop

```java
loop {
    Event ev = evQueue.remove()
    Handler handler = selectHandler(ev)
    execute(handler)
}
```

• The execution of asynchronous computation is atomic
  – events occurring while a handler is in execution are enqueued in the event queue
REMARKS

• Event handler are meant to be executed without blocking
  – no blocking primitive/call/request should be allowed
  – a blocking behaviour must be replaced by an asynchronous request or computation, which will eventually generate some event in the future
    • e.g. about the availability of some result or an error

• The event loop is embedded inside the runtime executing the code
  – developers don’t see it, don’t need to create loops: just specify how to select and execute event handlers
  – strong similarities with (stateful) actors event loop

• Event-driven programming is also called programming without a call stack
  – event handling is not a procedure call
    • there is no return point, there is no need of a stack
EVENTS VS. THREADS: AN OLD DEBATE

• Across different topics
  – Why threads are a bad idea (for most purposes). John Ousterhout - Invited Talk at the 1996 USENIX Technical Conference

• Each approach has pros and cons
  – the real challenge is about how to put them together…
BACK TO CALLBACKS AS CONTINUATIONS

• In this case:
  – events are implicit and refer to the completion with success or error of an asynchronous request/computations
  – event handlers are represented by continuation functions specified when invoking the asynchronous request/computation
  – closures are typically used to define the context to be used when processing the event
• Concrete examples - web app development
  – client side: Javascript, DART
  – server side: node.js
JAVASCRIPT - CLIENT EXAMPLE

- Event-loop ruling script execution
  - in web apps, events concern the web page life-cycle, the DOM, the user input, network interactions, … any event as defined by the HTML5 standard

- Example
  - Non blocking I/O
    ```javascript
    var ajaxRequest = new XMLHttpRequest;
    ajaxRequest.open('GET',url);
    ajaxRequest.send(null);
    ajax.onreadystatechange = function() {
      // what to do next …
    }
    ```

  - wrong way of doing it…
    ```javascript
    // this is wrong
    var ajaxRequest = new XMLHttpRequest;
    ajaxRequest.open('GET',url);
    ajaxRequest.send(null);
    while (ajaxRequest.readyState === XMLHttpRequest.UNSENT){}
    // !DEADLOCK!
    ```
JAVASCRIPT - CLIENT EXAMPLE

- `setTimeout(callback, timeout)` is another asynchronous function
  - registering a callback to be executed when the timeout expires
- What is the output of the following script?

```javascript
for (var i = 1; i <= 3; i++){
  console.log("setting timer..."+i);
  setTimeout(function(){ console.log(i); }, 0)
}
```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(exists) {
            fs.readFile(filename, function(err, data) {
                response.writeHead(200);
                response.end(data);
            });
        } else {
            response.writeHead(404);
            response.end();
        }
    });
}).listen(8080);

sys.log("Server running at http://localhost:8080/");
DART EXAMPLE

• DART language
  – https://www.dartlang.org
  – recent class-based language introduced by Google to develop large-structured web apps (client, server)
  – integrating Java+Javascript
    • can be translated into pure Javascript
  – “competitor”: Typescript, from Microsoft

• A Dart app has a single event loop with two queues
  – the event queue and the microtask queue
    • the event queue contains all outside events
      – I/O, mouse events, drawing events, timers, messages between Dart isolates, and so on.
    • the microtask queue contains events related to task to do, triggered from the program itself
      – it is necessary because event-handling code sometimes needs to complete a task later, but before returning control to the event loop.
While the event loop is executing tasks from the microtask queue, the event queue is stuck: the app can’t draw graphics, handle mouse clicks, react to I/O, and so on.
EVENT-DRIVEN PROGRAMMING: BENEFITS

- Using a single thread to coordinate and manage multiple asynchronous tasks
  - eventually executed by external/OS-backed threads
  - low memory footprint compared to multi-threaded solutions (where a stack is used for each thread)
- No low-level races
  - no shared state, only one thread accessing the state
- No low-level deadlocks
  - async handlers never block
PROBLEMS

• Besides the benefits, there are some main well-known problems affecting CPS and event-driven programming, often referred altogether as *callback hell*
  – **asynchronous spaghetti**
    • overall computation fragmented into asynchronous handlers
    • this tampers the modularity and structure of programs
      – making them difficult to understand and maintain
  – **pyramid of doom**
    • composition based on CPS leads to nested callbacks/continuations
    • this increases complexity and leads to poor source code readability, bad reusability, bad extensibility
PYRAMID OF DOOM

- Callback nesting

```
step1(function(result1){
    step2(function(result2){
        step3(function(result3){
            // and so on ...
        }
    }
}
```

“I love async, but I can’t code like this”
(a complaint of a developer on the Node.js Google Group)
PYRAMID OF DOOM

• In previous examples...

```javascript
function loadUserPic(userId, ret) {
    findUserById(userId, function(user) {
        loadPic(user.picId, ret);
    });
}

loadUserPic('john', function(pic) {
    ui.show(pic);
});
```

```javascript
// Node.js

var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(exists) {
            fs.readFile(filename, function(err, data) {
                response.writeHead(200);
                response.end(data);
            });
        } else {
            response.writeHead(404);
            response.end();
        }
    });
}).listen(8080);

sys.log("Server running at http://localhost:8080/");
```
A FURTHER EXAMPLE

- Taken from:

- It’s about a chat application, with three asynchronous calls:
  - the first one registers a user to a chat server
  - the second asks the server for an available room to join
  - and the third broadcasts an initial chat message to users in the room.

```javascript
registerToChatOnServer(username,function(rooms){
    joinAvailableRoom(rooms, function(roomname){
        sendChatToAll(roomname, msg, function(reply){
            showChatReply(reply);
        })
    })
});
```
A (PARTIAL) SOLUTION: PROMISES

• The callback hell can be partially solved by adopting the promise mechanism
  – originally proposed in 1976 by Daniel Friedman and D. Wise, as a proxy object that represents an unknown result that is yet to be computed (similar to futures)

• Promises represent the eventual completion and result of a single asynchronous operation
  – they encapsulate asynchronous actions, acting much like a value returned from the result of a computation only that the value may not be available at the time
  – a promise can be rejected or resolved once and only once
    • immutable once-resolved promises

• As a key property, they allow for flattening callback nesting
PROMISES IN JAVASCRIPT

- Defined in Javascript standard (*)
  - also called thenables
    - as a developer uses then to attach callbacks to a promise when it is either fulfilled or an exception is realised.
- Supported by different frameworks (with some different semantics...)
  - Q - Most complete & mature,
  - When.js - Lightweight
  - jQuery Promises (not fully conform)
- Example

```javascript
var promisedPic = loadUserPic('john');

 PROMISEDIC则 = loadUserPic('john');

promisedPic.then(function(pic) {
    ui.show(pic);
});
```

EXAMPLES

• Creating a promise

```javascript
var promise = new Promise(function(resolve, reject) {
    // do a thing, possibly async, then...

    if (/* everything turned out fine */) {
        resolve("Stuff worked!");
    }
    else {
        reject(Error("It broke"));
    }
});
```

• Attaching the callback

```javascript
promise.then(function(result) {
    console.log(result); // "Stuff worked!"
}, function(err) {
    console.log(err); // Error: "It broke"
});
```
PROMISE CHAINING

• “then” returns a promise, so that promises can be chained

```javascript
var promise = new Promise(function(resolve, reject) {
  resolve(1);
});

promise.then(function(val) {
  console.log(val); // 1
  return val + 2;
}).then(function(val) {
  console.log(val); // 3
});
```
PROMISE CHAINING

• Flattening the pyramid

```javascript
findUserById('john').then(function(user) {
    return findPic(user.picId).then(function(pic) {
        ui.show(pic);
    });
});

findUserById('john')
    .then(function(user) {
        return findPic(user.picId);
    })
    .then(function(pic) {
        ui.show(pic);
    });
```
PROMISE CHAINING IN Q

- Flattening the pyramid

```javascript
registerToChatOnServer(username, function(rooms){
    joinAvailableRoom(rooms, function(roomname){
        sendChatToAll(roomname, msg, function(reply){
            showChatReply(reply);
        }
    });
});

Q.fcall(registerToChatOnServer)
    .then(joinAvailableRoom)
    .then(sendChat)
    .then(function (reply){
        showChatReply(reply)
    },function (error){
        // catch error from all async operations
    })
    .done();
```
function get(url) {
// Return a new promise.
return new Promise(function(resolve, reject) {
// Do the usual XHR stuff
var req = new XMLHttpRequest();
req.open('GET', url);

req.onload = function() {
    // This is called even on 404 etc so check the status
    if (req.status == 200) {
        // Resolve the promise with the response text
        resolve(req.response);
    } else {
        // Otherwise reject with the status text which will hopefully be a meaningful error
        reject(Error(req.statusText));
    }
};
// Handle network errors
req.onerror = function() {
    reject(Error("Network Error"));
};
// Make the request
req.send();
}

get('story.json').then(function(response) {
    console.log("Success!", response);
}, function(error) {
    console.error("Failed!", error);
});
COMPOSING PROMISES

• In most libraries, promises are higher-level abstractions. So in addition to chaining, they can also be passed around and their operations composed easily, unlike asynchronous callbacks.
• A programmer can perform a number of high-level operations on promises such as *composing a group of independent asynchronous calls*
  – Fibonacci example using the Q library and the Q.all function:

```javascript
fibpromise = Q.all([ computeFibonacci(n-1), computeFibonacci(n-2) ]); 

fibpromise.spread(function (result1, result2) {
  //resolver for both operations..
  console.log(result1 + result2);
}, function(err){
  //error occurred
});
```

– To react to fibpromise we attach a resolver (with spread method)
  • the Q library provides Q.spread, which allows the arguments of the individual elements in a composition to be sent to the resolver
  • in this case, fibpromise awaits for both Fibonacci sequences to be computed before it passes the results to the resolving function which then sums the results
OTHER FRAMEWORKS

• Other frameworks providing the abstractions for implementation of promises in web app dev:
  – AngularJs
    • a JavaScript MVC-based framework for web applications
  – Dart
    • promises have been recently re-implemented as Dart futures

• Outside the web, promises are widely employed as abstractions in the domain of concurrent and distributed languages
  – distributed, actor-based language AmbientTalk uses similar concepts known as futures for non-blocking remote invocations, which can composed by grouping.
  – Argus provides promise pipelining, similar to the aforementioned promise chaining
  – Fignale provides future combinators to compose delayed actions in a scatter/gather pattern
PROMISES ARE NOT A SILVER BULLET

• Promises are a quite elegant solution to solve the nested-callback problem
  – providing a portable encapsulation mechanism capable of being passed around
  – afford programmers the convenience of handling them as first class objects
  – a great advantage given asynchronous mechanisms they represent
• But “not everything can be promised”
  – incremental processing
    • promises are for processing the full response
  – event processing & streams
    • UI logic
    • we need one promise for each event instance
  – temporal logic
• Concrete examples
  – Live search
  – Processing streams of pushed data (SSE, WebSockets...)
  – Incremental processing if large data sets as streams
REACTIVE PROGRAMMING

• Both the CPS and Promises assume that asynchronous computations will deliver their result in one shot.
  – however it is not uncommon in application the need to manage (asynchronous) **streams** of data/events.
  – this is the goal of **Reactive Programming**

• Reactive Programming paradigm
  – oriented around data flows and the propagation of change
  – easing the management of asynchronous streams of data and events
    • strongly related to the Observer pattern & event-driven programming
  – originally introduced in 1990s, it is attracting much attention today as an effective approach for developing responsive (web) apps, in particular in the context of Big data
**RP LANGUAGES AND FRAMEWORKS**

- *Reactive extensions* have been introduced for all major programming languages/systems
  - FrTime (based on Scheme)
  - Microsoft's Reactive Extensions (Rx)
  - Reactive JavaScript - **Flapjax**, Bacon.js, AngularJs
  - Reactive DART
  - Scala.React
  - ...

RP ABSTRACTIONS

- Reactive programming abstracts time-varying events for their consumption by a programmer.
  - the programmer can then define elements that will react upon each of these incoming time-varying events
  - abstractions are first-class values, and can be passed around or even composed within the program.

- Two main kinds of reactive abstractions:
  - **event streams**
    - modeling continuous and discrete (or sparse) time-varying values
    - asynchronous abstractions for the progressive flow of intermittent, sequential data coming from a recurring event
    - e.g. mouse events can be represented as event streams.
  - **behaviours (or signals)**
    - continuous values over time
    - abstractions that represent uninterrupted, fluid flow of incoming data from a steady event.
    - e.g. a timer can be represented as a behaviour.
EXAMPLE IN FLAPJAX

• A timer in Flapjax

```javascript
var timer = timerB(100);
var seconds = liftB(
    function (time){
        return Math.floor(time / 1000);
    }, timer);

insertDomB(seconds, 'timer-div');
```

• This examples set up a DOM element in web page showing a timer count in seconds.
  - first a timer behaviour producing a value after every 100 milliseconds is created (timer)
    • timer is a behaviour variable
  - to show the time in second, a further behaviour variable is defined (seconds), storing the flow in seconds
  - finally the "seconds" behaviour is inserted within the DOM element with the ID timer-div on a webpage
    • the value of the timer will therefore be continuously updated and displayed on the page
LIFTING

• The values in expressions that depend on the reactive values need to be reactive themselves.
• A variable which gets assigned with some expression involving behaviours or event streams becomes a reactive variable itself
  – since updates from the behaviours/event streams will affect the variable itself
• The process of converting a normal variable into a reactive variable is called lifting
• In order to correctly recompute all the reactive expressions once a event stream or behaviour is triggered, most libraries construct a dependency graph behind the scenes.
  – whenever an expression changes, the dependent expressions are recalculated and their values updated
  – In the timer example, a time change that happens updating timer triggers the re-evaluation of the function (time){...} and the update of seconds, which subsequently updates the value inserted in the web page
Implicit vs. explicit lifting

- a number of JavaScript libraries implicitly perform lifting for the programmer – e.g. in Bacon.js.
- for some, the programmer has to perform lifting of reactive expressions manually – such as in React.js
- a third category of libraries offer both implicit and explicit lifting – for instance Flapjax
- which if used as a library the programmer perform lifting explicitly but if used as a compiler the code is transformed implicitly
COMPOSING EVENT STREAMS AND BEHAVIOURS

• A essential feature in reactive programming is reactive abstraction composition
  – this allows to avoid the callback nightmare described previously
  – for instance, instead of having three separate callbacks to separately handle mouse clicks, mouse moves or mouse releases, we can compose them as a single event stream which responds to all the three events.

• Most JavaScript libraries provide this kind of fundamental support.
  – for example, in Bacon.js, properties (or behaviours) can be composed using the combine operator
  – in Flapjax, mergeE function is provided, which throttles the latest stream that comes in from either event.
COMPOSING EVENT STREAMS

- Live search example in Bacon.JS:

```javascript
var searchText = $('input[type="text"][name="search"]')
    .asEventStream('keyup')
    .map(".target.value")
    .map(trimText)
    .toProperty(''); // memory!

searchText
    .sample(500)
    .skipDuplicates()
    .filter(shorterThan(3))
    .map(searchForSearchTerm)
    .onValue(function (search) {
        search.on('data', ui.appendResult.bind(ui));
        search.on('end', ui.endSearching.bind(ui));
    });
```
COMPOSING EVENT STREAMS

• Example in Flapjax:

```javascript
var saveTimer = timerE(10000); //10 seconds
var saveClicked = extractEventE('save-button','click');
var save = mergeE(saveTimer,saveClicked);
save.mapE(doSave); //save received data
```

• In this example three event streams are created:
  – saveTimer is an event stream is created, that is triggered after every 10 seconds
  – saveClicked is an event stream that is triggered whenever a user clicks on the save-button
  – save is triggered whenever either the timer elapses or the save button is clicked
  – finally, for each element of the save stream, the doSave function is called (functional style)
REACTIVE EXTENSIONS

• An approach integrating aspects of reactive programming in mainstream languages is given by **Reactive Extensions (Rx)**
• Rx is a library for composing **asynchronous** and **event-based** programs using **observable collections**
• Three core properties:
  – **Asynchronous and event-based**
    • Rx’s mission statement is to simplify those programming models. A key aspect for developing reactive user interfaces, reactive web app, cloud app - where asynchrony is quintessential.
  – **Composition**
    • Simplifying in particular the composition of asynchronous computations
  – **Observable collections**
    • leveraging the active knowledge of programming model like LINQ, by looking at asynchronous computations as observable data sources
    • A mouse “becomes a database of mouse moves and clicks”: such asynchronous data sources are composed using various combinators in the LINQ sense, allowing things like filters, projections, joins, time-based operations, etc.
REACTIVE EXTENSIONS

• Rx has been originally introduced as a library on .NET
  – inspired to research work on functional reactive programming (FRP)
  – one of the hottest Microsoft Open Technologies today
    • https://rx.codeplex.com/
• Besides C# and .NET, nowadays "reactive extension" are being developed for almost every programming language
  – RxJS, RxJava, RxScala, RxPython..
  – 
• Main vision explained in the paper:
PRINCIPLES

• Using Rx, we can
  1. represent multiple asynchronous data streams coming from diverse sources
     • e.g., stock quote, tweets, computer events, web service requests, etc.
  2. subscribe to the event stream using the IObserver<T> interface (on .NET)
     • the IObservable<T> interface notifies the subscribed IObserver<T> interface whenever an event occurs.

• Because observable sequences are data streams, we can query them using standard LINQ query operators implemented by the Observable extension methods.
  – thus you can filter, project, aggregate, compose and perform time-based operations on multiple events easily by using these standard LINQ operators.

• In addition, there are a number of other reactive stream specific operators that allow powerful queries to be written.
  – cancellation, exceptions, and synchronization are also handled gracefully by using the extension methods provided by Rx
IOBSERVABLE

- Rx complements and interoperates smoothly with both synchronous data streams (IEnumerable<T>) and single-value asynchronous computations (Task<T>) as the following diagram shows:

<table>
<thead>
<tr>
<th></th>
<th>Single return value</th>
<th>Multiple return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull/Synchronous/Interactive</td>
<td>T</td>
<td>IEnumerable&lt;T&gt;</td>
</tr>
<tr>
<td>Push/Asynchronous/Reactive</td>
<td>Task&lt;T&gt;</td>
<td>IObservable&lt;T&gt;</td>
</tr>
</tbody>
</table>
interface IObserver<T> {
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}

interface IDisposable {
    void Dispose();
}
**SIMPLE EXAMPLE**

```csharp
IObservable<int> source = Observable.Range(1, 10);
IDisposable subscription = source.Subscribe(
    x => Console.WriteLine("OnNext: {0}", x),
    ex => Console.WriteLine("OnError: {0}", ex.Message),
    () => Console.WriteLine("OnCompleted"));
Console.WriteLine("Press ENTER to unsubscribe...");
Console.ReadLine();
subscription.Dispose();
```

- When an observer subscribes to an observable sequence, the thread calling the Subscribe method can be different from the thread in which the sequence runs till completion.
- Therefore, the Subscribe call is *asynchronous* in that the caller *is not blocked until the observation of the sequence completes*.
- This will be covered in more details in the Using Schedulers topic.
BRIDGING WITH EXISTING .NET EVENTS

var lbl = new Label();
var frm = new Form { Controls = { lbl } };
IObservable<EventPattern<MouseEventArgs>>
    move = Observable.FromEventPattern<MouseEventArgs>(frm, "MouseMove");
move.Subscribe(evt => {
    lbl.Text = evt.EventArgs.Location.ToString();
});

- The mouse-move event stream of a Windows form is converted into an observable sequence.
- every time a mouse-move event is fired, the subscriber will receive an OnNext notification.
- We can then examine the EventArgs value of such notification and get the location of the mouse-move
BRIDGING WITH EXISTING ASYNCHRONOUS SOURCES

Stream inputStream = Console.OpenStandardInput();
var read =
    Observable.FromAsyncPattern<byte[], int, int, int>(
        inputStream.BeginRead, inputStream.EndRead);
byte[] someBytes = new byte[10];
IObservable<int> source = read(someBytes, 0, 10);
IDisposable subscription = source.Subscribe(
    x => Console.WriteLine("OnNext: {0}".Format(x)),
    ex => Console.WriteLine("OnError: {0}".Format(ex.Message)),
    () => Console.WriteLine("OnCompleted"));
Console.ReadKey();

- BeginRead and EndRead for a Stream object which uses the IAsyncResult pattern are converted to a function that returns an observable sequence.
QUERYING OBSERVABLE SEQUENCES USING LINQ OPERATORS

- **Combining streams**

```csharp
class Program
{
    static void Main()
    {
        var source1 = Observable.Range(1, 3);
        var source2 = Observable.Range(1, 3);
        source1.Concat(source2)
            .Subscribe(Console.WriteLine);
    }
}
```

The resultant sequence is 1,2,3,1,2,3. This is because when you use the Concat operator, the 2nd sequence (source2) will not be active until after the 1st sequence (source1) has finished pushing all its values. It is only after source1 has completed, then source2 will start to push values to the resultant sequence. The subscriber will then get all the values from the resultant sequence.

- **Merge streams**

```csharp
class Program
{
    static void Main()
    {
        var source1 = Observable.Range(1, 3);
        var source2 = Observable.Range(1, 3);
        source1.Merge(source2)
            .Subscribe(Console.WriteLine);
    }
}
```

In this case, the resultant sequence will get 1,1,2,2,3,3. This is because the two sequences are active at the same time and values are pushed out as they occur in the sources. The resultant sequence only completes when the last source sequence has finished pushing values.

- **Projection**

```csharp
class Program
{
    static void Main()
    {
        var seqNum = Observable.Range(1, 5);
        var seqString =
            from n in seqNum
            select new string('*', (int)n);
        seqString.Subscribe(
            str => { Console.WriteLine(str); });
        Console.ReadKey();
    }
}
```

Mapping a stream of num into a stream of strings.
QUERYING OBSERVABLE SEQUENCES USING LINQ OPERATORS

• Other important and useful operators
  – filtering, time oriented operations, handling exceptions…
• In general:
  – **transform** data
    • using methods like aggregation and projection.
  – **compose** data
    • using methods like zip and selectMany.
  – **query** data
    • using methods like where and any.
RXJAVA

• RxJava is a Java VM implementation of Reactive Extensions
  – from Netflix
• It extends the observer pattern to support sequences of data/events and adds operators that allow you to compose sequences together declaratively
  – abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O
• It supports Java 5 or higher and JVM-based languages such as Groovy, Clojure, JRuby, Kotlin and Scala.
ASYNCHRONOUS PROGRAMMING: CHALLENGES

• Neither reactive programming or Reactive extensions can be considered the silver bullet for asynchronous programming
  – they are effective to manage asynchronous streams of data / events in a functional style…
  – ..but not as a general-purpose asynchronous programming model
• The challenge today is how to integrate all these approaches and techniques
  – synch + asynch, push + pull
  – exploiting concurrency
    • multiple communicating event loops (=> actors)
  – devising models that work also in the case of distributed systems
SUMMARY

• Asynchronous programming
  – based on future/task
  – based on CPS and callbacks
• Event-loop model
• callback hell
  – asynchronous spaghetti, callback nesting
• Promises
  – chaining, composition
• From events to event streams: Reactive programming
  – flapjax
• Reactive extensions (Rx)
  – Observable, observers