Asynchronous computations are lightweight software threads supported by the runtime that are more efficient than operating system (OS) threads, particularly when many threads are created or they often need to wait for events. They use continuations/callbacks to asynchronously wait on events, so that their OS thread can be reused for other computations.
Asynchronous Computations

- Asynchronous Computations (async’s) are an alternative to creating threads.
  - Creating OS threads and swapping between them is costly, both in time and space.
- In many cases we may only need a thread briefly, and then it waits on another event, so this is a waste.
- With async’s we instead define a set of computations, with a small number of actual threads swapping between them.
- The runtime allocates a threads – often 1-2 per core/CPU.
- When a computation needs to wait for some action to complete, instead of making the thread wait, it creates a continuation function that will be called by the runtime when the action completes.
- The continuation may be called in a different thread.
- This style of program is becoming increasingly common
  - e.g., AJAX for lightweight threads in a browser, Intels Threading Building Blocks, and many more.
- Programming with continuations directly can be awkward, so F# has an elegant workflow syntax that simplifies writing asynchronous computations.
  - This allows using a “let” rather than creating continuations explicitly.
  - Sequence (& list/array) comprehensions are actually a kind of workflow, and they can be used for a wide variety of purposes.
Asynchronous computations - example

```fsharp
open System.Net
open System.IO

let museums = ["MOMA", "http://moma.org/";
    "British Museum", "http://www.thebritishmuseum.ac.uk/";
    "Prado", "http://museoprado.mcu.es"]

let fetchAsync(nm,url:string) =
    async {
        do printfn "Creating request for %s..." nm
        let req = WebRequest.Create(url)
        let! resp = req.AsyncGetResponse()
        do printfn "Getting response stream for %s..." nm
        let stream = resp.GetResponseStream()
        do printfn "Reading response for %s..." nm
        let reader = new StreamReader(stream)
        let! html = reader.AsyncReadToEnd()
        do printfn "Read %d characters for %s..." html.Length nm
    }

for nm,url in museums do
    Async.Spawn(fetchAsync(nm,url))  // Runs concurrently
```

val museums : (string * string) list
val fetchAsync : string * string -> Async<unit>
Asynchronous computations are done concurrently:

Creating request for MOMA...
Creating request for British Museum...
Creating request for Prado...
Getting response for MOMA...
Reading response for MOMA...
Getting response for Prado...
Reading response for Prado...
Read 188 characters for Prado...
Read 41635 characters for MOMA...
Getting response for British Museum...
Reading response for British Museum...
Read 24341 characters for British Museum...
Asyncs – example with thread IDs

- If we modify the program to print the thread ID numbers by replacing each call to `printfn` with `tprintfn`:

```fsharp
let tprintfn fmt =
    printfn "[.NET Thread %d]"
        System.Threading.Thread.CurrentThread.ManagedThreadId
    printfn fmt
```

Then we find that each async hops between threads:

```
[.NET Thread 12]Creating request for MOMA...
[.NET Thread 13]Creating request for British Museum...
[.NET Thread 12]Creating request for Prado...
[.NET Thread 8]Getting response for MOMA...
[.NET Thread 8]Reading response for MOMA...
[.NET Thread 9]Getting response for Prado...
[.NET Thread 9]Reading response for Prado...
[.NET Thread 9]Read 188 characters for Prado...
[.NET Thread 8]Read 41635 characters for MOMA...
[.NET Thread 8]Getting response for British Museum...
[.NET Thread 8]Reading response for British Museum...
[.NET Thread 8]Read 24341 characters for British Museum...
```

- This is because OS threads are reused instead of blocking
Asynchronous computations in detail

- The type `Microsoft.FSharp.Control.Async<'a>` represents a computation that will generate an 'a value later.

- This type is implemented roughly as follows.

  ```
type Async<'a> = Async of ('a->unit) * (exn->unit) -> unit
  ```

- To run an async, two *continuations* must be provided:
  - `success:'a->unit` which is called with the generated value
  - `fail:exn->unit` which is called instead if the async fails

- Basic operations like `req.AsyncGetResponse()` return the thread to the .NET thread pool after arranging for the continuations to be called when the operation completes.

- Larger async’s are built from basic ones via the *workflow syntax*:
  ```
  async { ... }
  ```

- [An async can also be cancelled - see Ch13 of the recommended reading.]
Workflows

- Workflows are special syntax that can be used to compose different types of computations involving doing a sequence of things, but not a standard F# sequence of actions.
- They are based on the mathematical concept of a monad (\& on Haskell).

```fsharp
let async = async { let req = WebRequest.Create("http://moma.org/")
  let! resp = req.AsyncGetResponse()
  let stream = resp.GetResponseStream()
  let reader = new StreamReader(stream)
  let! html = reader.AsyncReadToEnd()
  return html }
```

is actually an abbreviation for:

```fsharp
let async = async.Delay(fun () ->
  let req = WebRequest.Create("http://moma.org/")
  async.Bind(req.AsyncGetResponse(), fun resp ->
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    async.Bind(reader.AsyncReadToEnd(), fun html ->
      async.Return(html) )) )
```

- Here `async` is a special object with members that indicate how the workflow constructs should be interpreted for this kind of workflow.
AsyncBuilder

- `async` is a instance of the following type/class (defined in Fsharp.Core)

```fsharp
type AsyncBuilder with
    member Return: 'a -> Async<'a>
    member Delay: (unit -> Async<'a>) -> Async<'a>
    member Using: 'a * ('a -> Async<'b>) -> Async<'b>
    member Bind: Async<'a> * ('a -> Async<'b>) -> Async<'b>
```

- **let** works like a standard let – synchronously
- **let!** executes an asynchronous computation, via continuations.
  - This happens via the `Bind` member.
  - The whole rest of the workflow is turned into a continuation function.
  - The pattern on the LHS of the `=` is used for the continuation’s argument
- **return** calls the success continuation with the given value.
- See the recommended reading Ch13 for more details.
## Constructs for asynchronous blocks

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>let! pat = expr</code></td>
<td>Execute the async computation <code>expr</code> and bind its result to <code>pat</code> when it completes. If <code>expr</code> has type <code>Async&lt;'a&gt;</code>, then <code>pat</code> has type <code>'a</code>. Equivalent to <code>async.Bind(expr, (fun pat -&gt; ...))</code>.</td>
</tr>
<tr>
<td><code>let pat = expr</code></td>
<td>Execute an expression synchronously and bind its result to <code>pat</code> immediately. If <code>expr</code> has type <code>'a</code>, then <code>pat</code> has type <code>'a</code>. Equivalent to <code>async.Let(expr, (fun pat -&gt; ...))</code>.</td>
</tr>
<tr>
<td><code>do! expr</code></td>
<td>Equivalent to <code>let! () = expr</code>.</td>
</tr>
<tr>
<td><code>do expr</code></td>
<td>Equivalent to <code>let! () = expr</code>.</td>
</tr>
<tr>
<td><code>return expr</code></td>
<td>Evaluate the expression, and return its value as the result of the containing asynchronous workflow. Equivalent to <code>async.Return(expr)</code>.</td>
</tr>
<tr>
<td><code>return! expr</code></td>
<td>Execute the expression as an asynchronous computation, and return its result as the overall result of the containing asynchronous workflow. Equivalent to <code>expr</code>.</td>
</tr>
<tr>
<td><code>use! pat = expr</code></td>
<td>Like <code>let!</code> but calls <code>Dispose</code> on each variable in the pattern when the enclosing async ends. Equivalent to <code>async.Bind(expr, (fun x -&gt; async.Using(x, fun pat -&gt; ...)))</code>.</td>
</tr>
<tr>
<td><code>use pat = expr</code></td>
<td>Like <code>let!</code> but calls <code>Dispose</code> on each variable in the pattern when the async ends. Equivalent to <code>async.Using(expr, (fun pat -&gt; ...))</code>.</td>
</tr>
</tbody>
</table>
## Async static members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async.Primitive: ('a -&gt; unit) * (exn -&gt; unit) -&gt; Async&lt;'a&gt;</td>
<td>Builds a primitive asynchronous step.</td>
</tr>
<tr>
<td>Async.Parallel: #seq&lt;Async&lt;'a&gt;[]&gt; -&gt; Async&lt;'a&gt;[]</td>
<td>Runs the async’s in parallel and collects the results. Cancels all when exceptions.</td>
</tr>
<tr>
<td>Async.Run: Async&lt;'a&gt; -&gt; 'a</td>
<td>Runs via the thread pool and waits for the result.</td>
</tr>
<tr>
<td>Async.Spawn: Async&lt;unit&gt; -&gt; unit</td>
<td>Queues without waiting.</td>
</tr>
<tr>
<td>Async.SpawnChild: Async&lt;unit&gt; -&gt; Async&lt;unit&gt;</td>
<td>As above, but inherits the cancellation handle from the current async.</td>
</tr>
<tr>
<td>Async.SpawnThenPostBack: Async&lt;'a&gt; * ('a -&gt; unit) -&gt; unit</td>
<td>As above, but then its result is available, executes the given callback.</td>
</tr>
<tr>
<td>Async.Catch: Async&lt;'a&gt; -&gt; Async&lt;Choice&lt;'a,exn&gt;&gt;</td>
<td>Catches errors from an asynchronous computation and returns a Choice result indicating success or failure.</td>
</tr>
</tbody>
</table>

Catch makes use of: type Choice<'a,'b> = Choice2_1 of 'a | Choice2_2 of ‘b